Vakgroep Zuivere Wiskunde en Computeralgebra

Towards the Correctness of Software Behavior in UML

A Model Checking Approach based on Slicing

Sara Van Langenhove

Promotor : Prof. Dr. A. Hoogewijs

Proefschrift voorgelegd aan de Faculteit Wetenschappen
tot het behalen van de graad van Doctor in de Wetenschappen: Informatica.

Preface

The importance of computer applications has grown quite a lot in the last few years. These applications influence and sometimes control many areas of human life. Nowadays, it is possible to find a programmed computer chip in many products starting with common things like a TV set or a mobile phone. The list continues with medical tools and ends with high-tech products like for example airplanes. Obviously, each application needs to be extremely reliable, since every encountered problem costs lots of money, trust of customers and sometimes even human lives. To avoid these extra costs, it is important to restrict, as much as possible, our limited ability to design and implement complex systems with sufficiently high degree of confidence in their correctness under all circumstances. The problem of *design validation* — ensuring the correctness of the design at the earliest stage possible — is therefore the major challenge in any system development process.

Design validation is the task of testing a design adequately to ensure that all parts will function within specifications in end-user applications and environments. While testing explores some of the possible behaviors and scenarios of the system, leaving open the question of whether the unexplored trajectories may contain the fatal bug, *formal verification* conducts an *exhaustive exploration* of all possible behaviors. Thus, when a design is declared correct by a formal verification method, it implies that all behaviors have been explored.

At the beginning, formal verification was introduced to help proving correctness of computer hardware but recently also communication protocols and computer software have received proper attention. However, the usage of verification methods varies in the different areas. While verification of hardware is a daily routine for all hardware developers today, the state of software verification is still unsatisfactory. Things are changing ....

In this thesis, we are concerned with the formal verification of designs of (real-time) embedded systems that may be specified hierarchically. In order to perform formal verification, the design is specified as a set of interacting systems; each described as a finite state transition system. There are two major approaches to formal verification: *theorem proving* and *model checking*. Theorem proving has the advantage of effectively verifying state models using axioms and proof rules but it has two main disadvantages. Firstly, it requires sufficient expertise in formal methods and secondly, if and when a failed proof occurs, the theorem prover does not present a counterexample. As a consequence, only a small number of research groups has been able to apply this technique to industrial-scale software. Model checking has appeared as a clear competitor to the traditional verification method, particulary in developing reliable software for concurrent systems. In model checking, a specification or implementation is described as a transition system and then some properties of the system — expressed as a logic formula — are checked. Usually logics that are used, are suitable to express properties like *liveness* (something desired will eventually happen) or *safety* (something bad never happens).

The main problem that formal verification tools have to face is the fact that transition systems tend to consume too much memory, more than today's computers can manage. This problem usually is called *state space explosion*. Thus, it is quite complicated to develop an efficient verification tool for embedded applications. In contrast, hardware devices don't need such large transition systems due to some regularities in their design.

Abstraction is probably the most important technique for reducing the state space in order to create safe, compact, finite-state models that are suitable for verification. The verification will run faster regardless of the particular implementation techniques it employs. Some abstraction techniques reduce the size of the state transition graphs while others find a mapping between the actual data values in the system and a small set of abstract data values. *Program slicing*, as yet another abstraction technique, removes variables from the system that do not influence the behavior to be checked.

Scope and Goals of the thesis

The main goal of the thesis addresses embedded software verification, together with the corresponding methodology, in order to minimise the risks of software failure. Additionally, a *simple-to-use* tool is under development that will help in the formal verification of the applications, and that reduces the state space explosion problem.

Before any checking can begin, one is thus confronted with the task of specifying and designing the embedded system of interest. The implementation of the system is not considered here. The modeling step is difficult since no universal method exists to model a system. However, it is crucial to the relevance of the results subsequently obtained. It becomes easier when the Unified Modeling Language is used. This language is a graphical notation for creating software application designs in an object oriented manner. When used within a methodology, several of the language models are useful to represent different views. These views are used within the developing process for means of documentation, communication and requirements capture and allow the abstraction from implementation details such as programming language and computer platform.

Thanks to ...

From experience I can tell you that these first pages of a PhD thesis are the most widely read pages of the entire publication. It is here where you hope to find out that you have meant something in the life of the PhD candidate.

For me, this PhD has been an amazing journey with lots of ups and downs. The key to getting me to this stage was the tremendous help and support that I received from some important people throughout that time. It is with great pleasure that I now have the opportunity to express my gratitude to all of them, since saying "thank you" to someone is as satisfying as having them say it you.

My family has been the cornerstone of my existence, literally. I know I have been a busy bee these last years and not been around as much as either you or I would like, but knowing that I have a loving family to always turn to is something that is irreplaceable. My family taught me to persevere (mostly through sports) and always refused to let me quit studying computer science during my first year at university (and this was a hell of a job for them). So, in some sense, receiving this PhD is partly due to them. *Mams en Paps, Liesbet en Tom, Speedy, bedankt!*

This PhD would not have been possible to complete without the help of my supervisor Bert Hoogewijs. He kept an eye on the progress of my work and, sometimes, he found some time for me in a schedule that can only be described as extremely full. He helped me a lot in getting things formal in a correct way, and going over some concepts with him again and again was always a distinct exhaustive pleasure. With an enormous patience, he taught me to talk, to think, and to write. Besides being a good supervisor, he was also an advisor in some other aspects of life. I owe him lots of gratitude ... and he probably does not realize how much he taught me.

I would also like to give special thanks to the members of my thesis

examination committee for their support and valuable comments: Jan Willem Klop, Dirk van Heule, Herman Tromp, Eric Laermans, and Kris Coolsaet.

Money is always important no matter what people say! This research project was generously supported by grants provided by the Ghent University (BOF/GOA project B/03667/01 IV1) and the Prof. Dr. Wuytack Fund. It was really wonderful to travel around the world and to meet all sorts of people.

Somewhere in the first half of this journey, I met Hans (soon to be my husband), who has been by my side day and night throughout the entirety of my PhD from that point on. Thank you for showering me with your love through all my moods, for your support, for your special cheering up ability, for the good times spent in awesome places, for distracting me, for sometimes giving me the time to work on my PhD, for finding stupidities in my manuscript, for being "the new man", ..., really thanks for everything. *Glad that my PhD has finally finished?*

I had the pleasure to supervise and work with several students who did their graduation work in our project and who have been somehow contributed to the work presented in this thesis. So a word of thanks to all of them. Ann Vanhoof (HoGent) did a nice job struggling with graphical user interfaces. The splendid construction of the CaSMV model for state charts gave Hans Hendrickx many sleepless nights. Benjamin De Leeuw and Tim Blanckaert tested two UML case tools; the one with more success than the other. For some years, they also have been my colleagues. I was really amazed how they so quickly turned out to be such silent neighbours at work.

Working as a computer scientist in a department full of mathematicians is an extraordinary experience, but I will leave open what this exactly means. Thanks to all for everything.

In the end, I'm very happy to have done this PhD research. You learn a lot. Doing individual research. Figuring out what needs to be done. Writing. Writing a lot. Writing an entire book even. Learning to be critical of things (papers, research, technologies), something that didn't always come naturally to me. Doing a PhD is a combination of very fun and interesting work, lots of flexibility and freedom, but at the price of some really hard work.

Cheers Everybody!

Sara Van Langenhove
Gent, 4 Mei 2006

Contents

I Verification

CHAPTER 1

Introduction

The fearful unbelief is unbelief in yourself.
Thomas Carlyle.

In recent years, the use of embedded software has risen to a previously unseen level. Everywhere we see devices that utilize modern computer technology to improve usability or performance; cars, telephones, vending machines etc. all use embedded software nowadays.

As we all know from our computers, there are bugs in all useful computer programs. While bugs are irritating, they are often harmless on our home PC but not accepted in embedded software; if the brake system of a car fails it is too late to submit a bug report.

Embedded systems are often safety critical and therefore require high-quality design and guaranteed properties. Model-based design has been advocated as the method of choice for dealing with systems as these. The design process consists of building models on which the required properties are carefully checked before continuining with implementation phases. This allows high quality to be achieved at lower costs.

We will start the introduction by defining embedded systems. Then, the importance of designing is considered together with a short description of a succesful design language. To continue, the need for verification is explained in great detail together with a verification method (and its optimization) that is particularly used throughout the thesis. Finally, an outline of the thesis is given to briefly show what we have considered to achieve high quality of embedded systems.

1.1 Embedded Systems

Deep in the intellectual roots of computation is the notion that software is the realization of mathematical functions as procedures. These functions map a body of input data into a body of output data. Such functions are widely used in various areas for all kinds of applications available for various needs, such as spreadsheets, word processors, scientific calculators and tools, etc.

Embedded software is not like that since they are mostly control-oriented rather than data-oriented. Embedded computers [75] are processing devices used in diverse areas as wireless communications, medical instrumentation, food preparation, water treatment facilities and so on. Users of these systems may not be aware of the CPU embedded within, making decisions about how and when the system should act, as illustrated in Example 1.1 The users are not intimately involved with such a device as a computer *per se*, but rather as an electrical or mechanical appliance.

Example 1.1. *Consider a simple pumping control system [98] that transfers water from a source tank A into another sink tank B using a pump, as shown in Figure 1.1. Each tank has two meters to detect whether its level is empty or full. The level is ok if it is neither empty or full. Controls on the water levels are used to switch the pump on or off i.e. the pump is switched off as soon as either tank A becomes empty or tank B becomes full.*

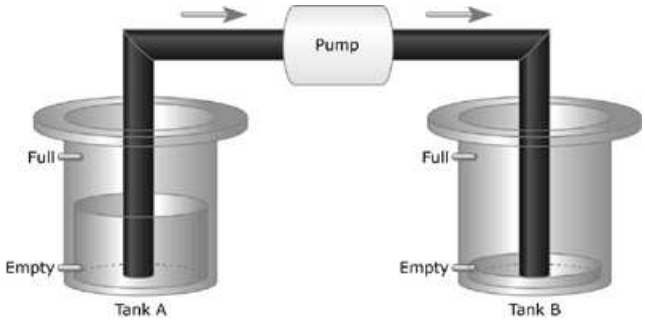


Figure 1.1: A Pumping Control System

Embedded systems [1] do not provide standard computing services and normally exist as part of a bigger system. A computerized washing-machine is an embedded system where the main system provides the feature of washing clothes with the help of an embedded computer. Embedded systems were

incorporated into commercial automobiles around the 1970s. The anti-lock braking system (ABS) is probably the best known. Today this trend continues with more innovative technologies such as adaptive cruise control and smart radar systems to avoid collisions. In each case, the embedded computer is part of a larger system that provides some noncomputing feature to the user. Usually, embedded systems are built with the least powerful computers that can meet both functional and performance requirements[1]. Of course, this is to reduce the manufacturing cost of the equipment but this does not mean that it is just software on small computers.

In conventional operating systems, a programmer easily allocates big chunks of memory without having to think about the consequences. These systems have enough main memory and a large pool of virtual memory to support such allocations. In contrast, the embedded system developers have to manage with complex algorithms to manage resources in the most optimized manner.

Embedded software is different from traditional software in many aspects [74]. It has additional properties like liveness, concurrency, reactivity, heterogeneity, and interfaces. Of course, all these properties are essential to the correctness of a program meaning that it is not sufficient to realize the right mapping from input data to output data. *Liveness* means that typical software in embedded systems does not terminate (unless it fails) or blocks waiting for events that will never occur. *Concurrent* processing is a feature of most embedded systems. This means that there are many events that need to be processed in parallel, and frequently, the order of incoming events is not predictable. Additionally, many embedded systems are *reactive* systems. They are event-driven and must respond to external stimuli. It is usually the case in reactive systems that the response made by the system to an input stimulus is state dependent i.e. the response not only depends on the stimulus itself but also on what has previously happened in the system. Naturally, embedded systems are a mixture of hardware and software components; the embedded software interacts with the hardware. The systems can also be *heteregoneous* in terms of different communication protocols or scheduling policies. *Interfaces* are universally accepted solutions to combine hardware and software after separate and independent development cycles. Its use enables the portability of software onto different hardware platforms together with the reuse of major parts of the software or hardware.

In most of the real-life applications, real-time systems often work in

[1]Functional requirements specify what should be done whereas performance requirements specify how well something has to be done. Every performance requirement presupposes a functional requirement, since specifying how well something should be done presupposes specifying what should be done as well.

an embedded context and most of the embedded systems have real-time processing needs. Such systems are called Real-Time Embedded Software Systems.

Real-Time Embedded Systems

The common feature of all real-time software systems [3, 38] is *timeliness* i.e. the requirement to respond correctly to inputs with acceptable time intervals. They encompass all devices with performance constraints. As defined by D. Gillies:

> *A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time in which the result is produced.*

A *hard* real-time system is a system whose operation is correct if results are produced according to the timing specification. A late response results into an erroneous computation and a system failure. In these systems, *late* data is *bad* data.

Soft real-time systems (e.g. airline reservation systems) are systems whose operation is degraded if results are not produced following the specified timing requirements. It is possible that the timing constraint is missed occasionally or missed by small deviations, or occasionally skipped altogether. Here, *late* data may still be *good* data.

Firm real-time systems combine both hard and soft timeliness requirements. The computation has a shorter soft requirement and a longer hard requirement.

1.2 The Need for Design

Many embedded systems interact with multiple sources at the same time. Such sources can be electrical devices, mechanical ones or just human beings. Frequently, custom software has to be specifically written to control the embedded system.

A first problem that arises with interaction is that the environment disregards the opinion of the developers of how and when the system ought to behave [38]. Another problem programmers of such applications have to face, is that they have to make sure that the application reacts to external events at the moment they occur, rather than when it might be convenient. Moreover, the embedded system developers have to implement complex algorithms on less hardware resources. The list of bottlenecks can easily be extended.

All the above reasons make embedded systems more complex and expensive to develop and to maintain. Developers often find themselves struggling to understand and to control them [8, 42, 131]. The question now is whether there exists a feasible way to construct embedded systems. To answer this question, let's first take a look at the following anecdote [103]:

> *When my son was 13, my husband and I thought he should have some space of his own. So we called in Leon the handyman and sat around the kitchen table. "What do you want?" he asked. Simple . . . a few closets, some electrical outlets, a cable hookup. Leon went off and remodeled the basement, and life was good. That is until winter came along and poor Michael comes upstairs and says, "Mom, I'm cold." You see, we hadn't thought about putting heat in the basement. So we decided to put in a gas fireplace. However, we don't have natural gas where I live, so we had to get a propane tank. Unfortunately, the only place it would fit was over in one corner, with all the storage in the other corner, and everything else in a third corner. The result is that you can't sit in my basement and see the TV and the fireplace at the same time. If we'd done some planning, we would have been able to design a room that was much more functional and comfortable.*

Clearly, the anecdote shows that software development, either embedded or not, is more than just typing in some lines of code. We must produce systems that scale as the underlying business grows and evolves at high speed. And these systems have to cope with the users' never ending appetite for functionality, scalability, extensibility, and reliability.

Nowadays, most programming languages provide lots of features to be capable to meet such users' demands [8, 131]. Consider for example the programming language *Java*[2]. Its associated virtual intermediate machine architecture provides platform independence by hiding the organizational details of the platform it runs on. Also, Java has an excellent support for exception handling and concurrency. Moreover, it is an advanced object-oriented language that allows the construction of very complex applications. Furthermore, object-oriented design is the key to both software reuse and the harnessing of complexity.

Unfortunately, all these features are not enough to develop very complex systems. Let's finish the anecdote [103]:

> *Now in this instance we made some mistakes, but it wasn't a serious disaster. Think about building a skyscraper, though. No one would dream of a major construction project without thorough blueprints.*

[2] Available from `http://java.sun.com/`

> *That's blueprints plural, because it's important to not only have one plan of the skyscraper — you need to have multiple plans. The electrician needs a view to show where the wiring goes. The plumber needs another plan so he doesn't put a sink in the elevator. And the carpenters need to know where to put this expensive crown molding in the CEO's office. Different workers need different views of what they're trying to build. And that's what we're doing with software.*

Embedded systems are systems in which the number, diversity of devices, amount of software, and degree of connectivity continuously increases. Therefore, they are never constructed by a single developer, an aspect the anecdote also adresses. Of course, developing embedded systems, or other complex systems, requires a clear understanding of the problem and a clear plan for the solution. In order to develop such understandings, developers need to visualize the system and communicate their decisions and creations to a wide audience. The conclusion is the same as the anecdote i.e. it is essential to have a design on which to base our efforts to develop such systems [8, 42, 131].

1.3 What is Design?

The heart of building (embedded) software systems is the construction of a design. In [131] a nice description is given:

> *What is design? ... It's where you stand with a foot in two worlds — the world of technology and the world of people and human purposes — and you try to bring the two together.*

A design is an attempt to fully understand the problem and to have a clear plan for the solution. It abstracts the essential details of the underlying problem from its usually complicated real world. It is the architectural model of a system that is created, modified and analyzed during the development cycle of a system [8, 42, 131]. It is the process of conceiving, inventing and contriving a scheme to be able to turn a specification for a (complex) system into an operational one. Design encompasses all the activities involved in conceptualizing, framing, implementing, commissioning and ultimately modifying complex systems [42].

The goal of design [8] is to make the next step, *implementation*, as simple and efficient as possible. During design, developers create a model of the objects that interact with each other to fulfil the requirements of a system. Design describes the solution, in great detail. Moreover, omitting design

is expensive and risky since it is cheaper and less time-consuming to make decisions at design level than at code level i.e. it prevents painful rework later [4, 8] (see section 1.6).

A design, also called a model, is far more accessible than the actual code that makes up the final system. It is a simplification of a reality; it is a blueprint of the actual system that needs to be built. Of course, a design is a visual model that enables more developers to understand, to (re)build, to evaluate and to review more of the system. This allows them to understand their own responsabilities and to estimate their effort more accurately. Within a design, decisions can be changed very quickly and efficiently. This contributes to the overall success of the project.

1.3.1 Different Views

Clearly, the anecdote indicates that a single model of a complex system has to describe the system from different perspectives [8, 103]. Each perspective is needed for a full understanding of the system. One view might show the system from the user's perspective while another view might describe the communication between several components of the system. Yet another view can be used to specify the behavioral part of some components.

It is important that the different views are visualized using a common language that is shared by all persons who are involved in the development process. Naturally, this requires a consistent use and understanding of the language. Moreover, the language must have plenty of features to be able to create several system perspectives. The *Unified Modeling Language* is such a language.

1.4 The Unified Modeling Language

In this thesis, we want to be able to cope with embedded systems which are given as models in the formalism of the Unified Modeling Language (UML) [96]. The Object Management Group (OMG) states:

> *The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.*

The important point to note here is that UML is a *language* intended to specify designs and not a method or procedure. The UML is used to define a software (embedded) system; to detail the artifacts in the system, to document and construct; it is the language that the blueprint is written in. The language is capable to separate business logic from the underlying platform technology. The UML may be used in a variety of ways to support a software development methodology but in itself it does not specify that methodology or process i.e. it does not tell you what to do first and what to do next or how to design your system.

1.4.1 History

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented (OO) analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." To stop these wars, the UML was constructed.

UML was a combination of three different approaches to modeling software. Ivar Jacobson had a methodology for viewing requirements called Use Cases that included diagrams very suspiciously similar to telecommunications diagrams.

Jacobson in 1992 more formally defined a method for modeling objects called Objectory. Around the same time James Rumbaugh was working on another method for modeling object-oriented systems called OMT. Yet another modeling method named after its creator, Grady Booch, was being developed and defined a couple of years later.

Rumbaugh and Booch started working together at Rational Software (now IBM Rational) [57] in 1994 to combine their two methods. Put into a draft in 1995, it became the now-famous 0.8 version of UML. Jacobson soon joined with Rumbaugh and Booch to bring his ideas for Use Cases and his experience into the mix. It was also around this time (in 1996) that the OMG [responsible for the CORBA standard] submitted a RFP (request for a proposal) for a standard method to model object-orientated software. By September of 1997 UML was submitted to the OMG, and was later approved in November of the same year. From that point forward, UML has been the standard modeling language for the OMG [96] and is, as of this writing, the defacto standard for modeling software. In 2004 the second version of UML was developed by the OMG in the form of UML 2.0. Even though the OMG controls the further development of the UML standard Booch, Rumbaugh,

and Jacobson will forever be remembered as the "Three Amigos" of software for their contribution to UML.

1.4.2 The Basics

UML is composed of many model elements [38, 96, 105] that represent different parts of a (embedded) software system. UML offers nine different diagram types for specifying both structure and behavior of a system. These are *use case diagrams* for catching the requirements of a system, *class diagrams* and *object diagrams* for describing its static structure, and *component diagrams* and *deployment diagrams* which depict its implementation structure. *Collaboration diagrams*, *sequence diagrams*, *statechart diagrams*, and *activity diagrams* specify the different aspects of behavior of a system, building up on the static structure defined in the corresponding diagrams described above.

All in all, UML tries to specify and visualize all aspects of software systems utilizing all nine diagram types. Each diagram type focuses on a specific aspect of the system to be built. Therefore, they are independent views of the model just as a number of computer screens looking into different records or parts of a database showing different views. Use case diagrams show the functionality of the system from an outside-in viewpoint. They are helpful to determine the required features the system is supposed to have. More precisely, they tell *what* the system should do but they cannot specify *how* this should be achieved. Class diagrams describe the structure and the interdependencies of the classes in an object-oriented system whereas object diagrams depict the instances of these classes. The latter ones can be used to test class diagrams for accuracy. In contrast, the interdependencies between physical pieces of software (e.g. in makefiles) are visualized by component diagrams and the relationships between software and hardware by deployment diagrams. Behavior that occurs between objects is grasped in different ways, either with focus on the structural dependencies between them (collaboration diagrams) or with focus on the message flow (sequence diagrams). The intraobject behavior is captured by statechart diagrams, whereas the workflow and other activities in the system are depicted with the help of activity diagrams.

Example 1.2. *The pumping control system of Example 1.1 easily extends to a controller for a complex network of pumps and pipes to control multiple source and sink tanks, such as those in water treatment facilities or chemical production plants. Now, let's add additional control changes to this embedded system. The pump is to be switched on as soon as the water level in tank A*

reaches ok, provided that tank B is not full. The pump remains on as long as tank A is not empty and as long as tank B becomes full. Such control changes can be visualized using statechart diagrams, as shown in Figure 1.2.

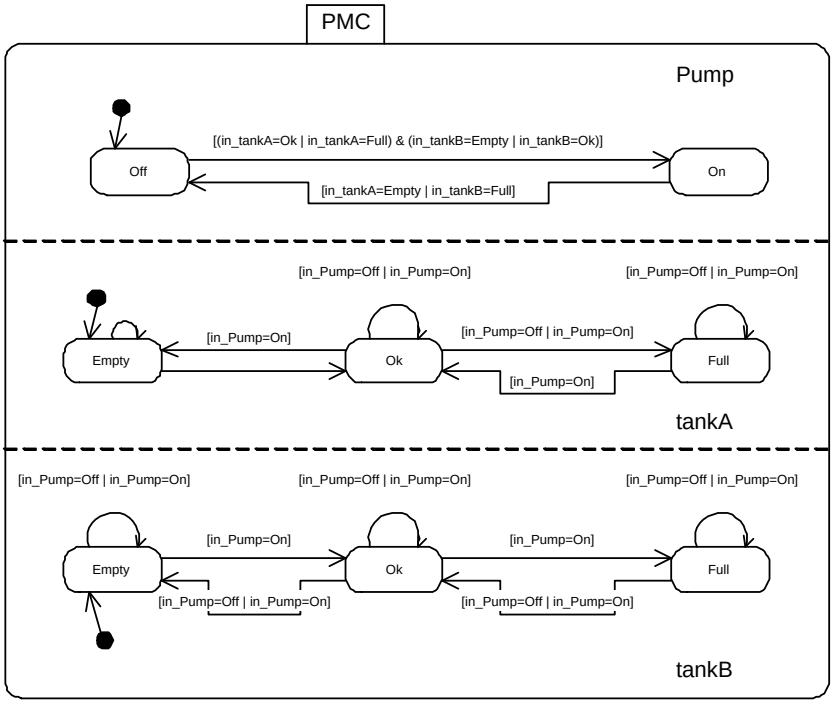


Figure 1.2: Statechart Diagram of a Pumping Control System

1.4.3 Tool Support

Over the years, UML has been established as a standard in object-oriented software engineering, defining parts and making systems more easily understandable. There exists a lot of UML-tools (either commercial or not) that are widely applied in the development of a wide range of systems. And most of them ultimately produce full production code from the model for the target system. We will mention a few of them.

Rational Rose [57], one of the best known UML-tools; free to universities, but not as fully featured as some of the others e.g. they do not support code generation for statecharts. Note that Rational was bought by IBM.

Rhapsody [58], is meant to be a clear competitor to Rational Rose, supporting model-driven development and all those good things. It is the tool, I have worked with during the last four years. Also, the company is member of the OMG.

A free UML tool is for example *ArgoUML* [111], but it runs far behind the others in the sense that they are slow in adapting to new versions of the UML standard. Another free one (for now) is *Jude* [29]. It looks like the authors intend to create a "plus" version for which they will charge money, while keeping the basic (fairly fully featured) version free.

1.5 The Need for Verification

Today, embedded systems are widely used in applications where failure is undesirable or even unacceptable: coffee-machines, medical instruments, air traffic control systems, video-on-demand applications, and other examples too numerous to list. We frequently read of incidents where some failure is caused by an error in a hardware or software system. A recent example of such a failure is the destruction of the Ariane 5 rocket [80], due to a floating point overflow; one bug and one crash [45].

The story of the Ariane 5 rocket tells us that the need for reliable hardware and software systems is critical [45, 71, 80]. As the involvement of such systems in our lives increases, so too does the necessity for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety; we are very much dependent on such systems for both their continuous operation and proper functioning [26]. Therefore, verification becomes important.

Verification will play an important role in each development process[3] to ensure that the system conforms to its specification and to guarantee that the system is functioning as expected by the users [4, 123]. Verification ensures that the developed system is correct and that the right system is developed. Naturally, with a product quality below expectations, users will definitely find a substitute system that better satisfies their needs. Suppose, for example, that the coffee-machine has software problems all the time, would you consider tea after a while? Of course, not only verified software is neccesary to avoid problems: if the coffee-machine suffers from regular hardware failure, then again one would consider tea after a while. When printing, the same mechanisms appear. If there are five networkprinters available, then people choose the best one. Again, people start looking for

[3] We concentrate our verification purposes only on the software part of embedded systems.

alternatives if those printers regularly fail due to soft- or hardware problems or due to the lack of local support.

Roughly speaking, each development process [107] starts with a requirement analysis that is followed by proper design phases that make the implementation of the system a lot easier. Of course, in each phase, possibly several methodologies are used. For the specification of (embedded) systems, usually several UML models representing different views are developed. In spite of the use of design languages like UML, the software design becomes more and more complicated so that the integrity of a system design, even in the early stages, is very difficult to be guaranteed. Furthermore, because UML is a very expressive and rich language, sometimes the model gives rise to behaviors that are not expected by the designers and those behaviors could cause serious bugs for the system.

Ideally, faults (inconsistencies, misconceptions, deadlocks, etc.) in a system have to be discovered as early as possible, to avoid painful rework later on in the development process [4]. Therefore, detailed designs are important since they provide the last chance to validate the solution before the expensive implementation process and test phases begin. Costs to detect and correct faults grow dramatically when these have propagated to later phases [2, 4]. Recovering from faulty reasoning *during* or *after* the next phases is a very tedious practice and incredibly expensive to fix. Observations have shown that the costs of correcting a fault in design and code phases is often 10 to 100 times less expensive than if it is found during the test phases [2, 4]. This is a major driver for focusing the effort on detecting faults early on in the development process. The goal of this thesis is to develop a method where faults may be detected during the design phase.

1.6 Our UML Design Verification Method

Do we want a *fast design* or a *correct design*? We all know the right answer by now. Design verification is becomming the major part of the design phase; its goal is to find faults in the design itself.

Some people still claim and still believe that the use of *formal methods is not required* [20]. We all know that this is a mistruth. Formal methods are mathematically-based approaches. They are highly important in systems where the issue of correctness is of concern. Formal methods have the potential to help eliminate errors early in the design process. Formal methods ensure that errors are not introduced into the development process [20, 21] because it is a tedious practice to remove errors during or after the code phase. The use of formal methods in the design phase improves the quality of

the software, whether or not mathematical approaches are used in subsequent phases of software development [21]. You may know that design verification is a formal verification technique that, of course, extensively uses formal methods.

Design verification is not the same as design validation. The problem of *design validation* — ensuring the correctness of the design at the earliest stage possible — can be solved by using techniques like *simulation* and *testing* [26]. Both methods typically inject signals at certain points in the system and observe the resulting signals at other points. These methods can be an efficient way to find many errors. However, checking *all* of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible, but extremely important when verifying embedded systems for their continuous operation, proper functioning in different circumstances, and so on. In contrast, *formal verification* such as design verification based on model checking [26], the main method used throughout this thesis, conducts an *exhaustive exploration* of all possible behaviors.

1.6.1 A Part of the Design

The previous sections have illustrated that for the specification of object-oriented (embedded) systems, usually several models representing different views are developed, independent of the used modeling language. Building models that faithfully represent complex systems is a non trivial problem and a prerequisite to the application of formal verification techniques even at a high level of abstraction. Since there are lots of things of a design (specified with UML) that can be verified, the design verification is here limited to the verification of one of the most important UML diagrams: the statechart diagram; i.e. we will focus on the verification of the behavior of objects. So why this limitation?

Typically, most embedded systems are control-oriented rather than data-oriented, meaning that the dynamic behavior is much more important than business logic (the structure and operations on the internal data maintained by the system). For control-oriented systems, UML statecharts [38, 96, 105] are widely accepted as good, clean and abstract representations to represent the dynamic behavior of classes of these systems.

Embedded systems do often have complex control schemes. They are characterized by concurrency aspects, by the synchronization and the communication among various entities inside or outside the system. Consequently, embedded statechart models become highly complex and as a result, error conditions easily find their way to inflitrate in these models. Moreover, it quickly becomes impossible to manually verify whether the

system possibly deadlocks, whether its components react on time, and so on and so forth. Properties like the mentioned ones are highly essential for the behavior of embedded systems and their correctness. Embedded system behavior must be free from any failure. That is the reason why this manuscript limits the verification of the design to the verification of UML statechart models.

1.6.2 The Process

It is one of the intentions of all the chapters to explain the verification methodology in detail and how it is step-wise built. Therefore, we will now just briefly give an overview; the black box of the verification process.

Figure 1.3 presents a verification process of embedded systems, whose behavior is expressed using UML statechart diagrams. These diagrams (saved using the XMI exchange syntax) are first transformed into an equivalent format, called Extended Hierarchical Automata [46, 68, 72, 124]. From these formats, a model in the language of the model checker Cadence SMV (CaSMV) [86] is generated. Additionally, behavioral temporal properties are injected into the model as well. Once having the complete model, which in fact is the combination of a finite state machine and a logical formula, the essential behavioral properties are automatically verified using the model checking technique. If the system does not satisfy a property, a counterexample (i.e. a path in the statecharts that does not fullfil the requirement) is returned. The software developers interpret the counterexample and fix the fault in the UML model; at that moment, the whole process of designing and verifying is restarted.

Details about the model checking technique are given in Section 1.6.4, which also motivates the use of it in our verification approach. The motivation for using CaSMV is given in Section 1.6.5. Another aspect of our UML verification methodology overcomes an important key obstacle of model checking: the state explosion problem. To improve the efficiency of model checking by tackling the state explosion problem, the technique of program slicing can be applied to the statechart diagrams that are part of the design. This problem is briefly discussed in Section 1.6.7.

One of the advantages of the method is that it can be fully automated. Another advantage is that the entire background mathematics of model checking is completely hidden to the software designers. This eliminates a specific expertise in the theoretical field of model checking.

In fact, we have built a Java-application [53, 122] that does the necessary transformations, calls the model checker at the appropriate time, and interprets the returned counterexample (the process visualized in Figure 1.3

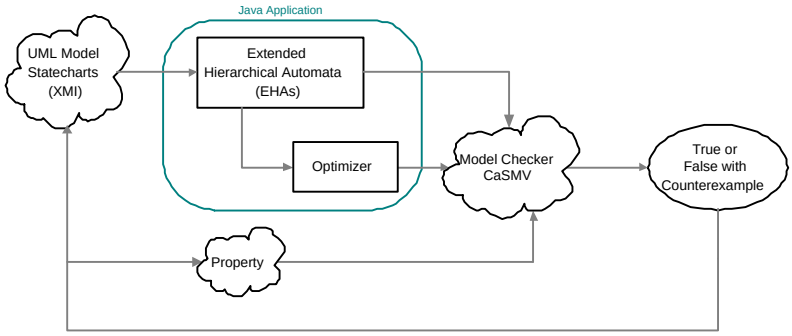


Figure 1.3: Black Box of Methodology & Tool Architecture

refers to the architecture of our Java application). Moreover, this application can be used as a plug-in tool for existing UML Case tools, like for example Rhapsody [58]. Additionally, the verification procedure can be used in an incremental design process. This means that each new version of the design can be automatically verified. Having an automated verification method, which is based on a mathematical approach and that easily integrates in the design phase, increases both our confidence in the UML system design and the use of it in industrial applications; at least, that is what we hope.

1.6.3 Model Checking

Model checking is the most successful approach that has emerged for verifying requirements of software and reactive systems. The essential idea behind model checking is shown in Figure 1.4. A model-checking tool accepts a system design (called *models*) and a property (called *specification*) that the final system is expected to satisfy. The tool then outputs *yes* if the given model satisfies given specifications and generates a *counterexample* otherwise. The counterexample details why the model doesn't satisfy the specification. By studying the counterexample, you can pinpoint the source of the error in the model, correct the model, and try again. The idea is that by ensuring that the model satisfies enough system properties, we increase our confidence in the correctness of the model.

Model: Kripke Structure

The model, used by a model checking tool, is a type of state transition graph, called a *Kripke structure* [26], to capture the behavior of embedded systems.

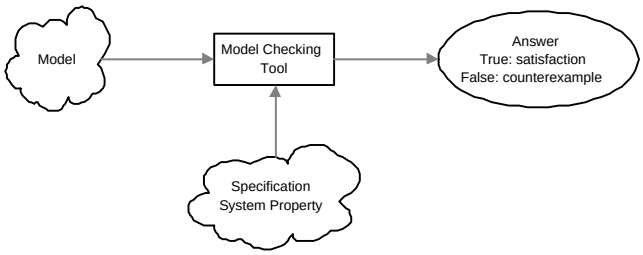


Figure 1.4: Model Checking Approach

Each *state* in a Kripke structure is essentially a tuple containing one value for each state variable i.e. a state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time. A *transition* in a Kripke structure denotes change in the value of one or more variables i.e. it shows how the state of the system changes over time. The *computations* of the model can be defined in terms of its transitions. A computation is an infinite sequence[4] of states where each state is reached from the previous state by some transition. These computations can be visually represented into an infinite tree where the root is labeled with our chosen initial state and the children of any state denote the next possible states. Obviously each path in the tree indicates a possible execution or behavior of the system.

Example 1.3. *Let's recap Example 1.1 and 1.2 again. Suppose that initially both tanks are empty and that the pump is turned off. The infinite execution tree is shown in Figure 1.5. Remember that a system state, represented by a bullet in the figure, is defined by a tuple of values for each of the three components in the system. For example,* $(tankA = Empty, tankB = Ok, pump = Off)$ *and* $(tankA = Empty, tankB = Full, pump = On)$ *are possible system states.*

Specification: Temporal Logic

The aim of model checking is to examine whether or not the execution tree satisfies a user-given property specification. The question now is how do we specify properties of paths (and states in the paths). Referring to Example 1.1, how do we state properties like *it is always possible to reach a*

[4]The infiniteness comes from the fact that each state must always have a successor state.

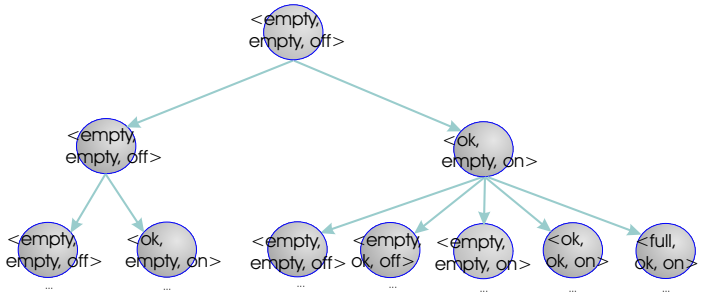


Figure 1.5: Execution Tree of a Pumping Control System

state where tank B is ok or full? In fact, *temporal logic* [26] is a simple and intuitive notation suitable for this purpose.

Temporal logic is a form of logic specifically intended for statements and reasonings which involve the notion order in time. It is a formalism for describing sequences of transitions between states in a system. Time, however, is not explicitly mentioned, but instead, a formula might specify that *eventually* some state is reached, or that an error state is never entered. Properties like *eventually, never or always* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily to build up complicated expressions describing properties. Temporal logics differ in the operators that they provide and the semantics of those operators. We will focus on *Lineair Temporal Logic* and *Computation Tree Logic* [12, 26].

The Lineair Temporal Logic: LTL Lineair temporal logic, which states path properties, studies the evolution of a system (through time) by examining the state sequence in a system's path. An LTL specification describes the intended behavior of a system on all possible executions. The logic is called linear since a system in a given state is only considered to have a single successor state in the next instant. The logic is the propositional logic built up from the elementary propositions augmented with five new operators (Figure 1.6, Figure 1.7):

- The unary operator **X** ("next time") requires that a property holds in the second state of the path.
- The unary operator **F** ("eventually" or "in the future") is used to assert that a property will hold at some state in the path.

- The unary operator **G** ("globally" or "always") specifies that a property holds at every state on the path.
- The binary operator **U** ("until"): φ_1 **U** φ_2 states that φ_1 is verified until φ_2 is verified.
- The binary operator **R** ("release") is the logical dual of the previous operator (φ_1 **R** φ_2). It requires that φ_2 holds along the path up to and including the first state where φ_1 holds, if φ_2 ever stops to hold i.e. the first property is not required to hold eventually.

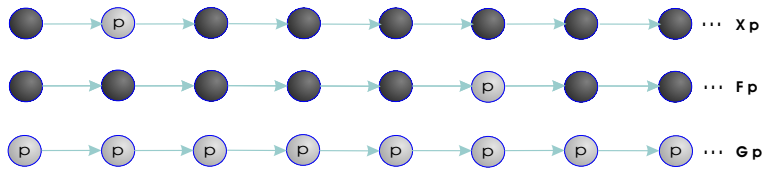


Figure 1.6: Unary LTL Operators

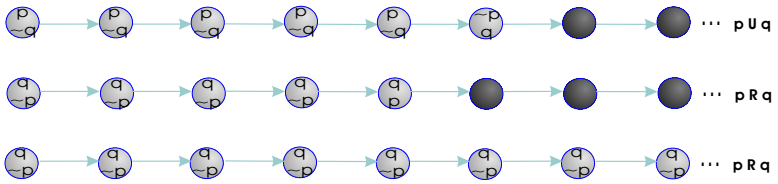


Figure 1.7: Binary LTL Operators

The Computation Tree Logic: CTL Computation tree logic expresses state properties that can take into account the branching structure of a transition system, i.e. that a state can have various distinct successors. For instance, many futures are possible starting from a given state. Special purpose path quantifiers, **A** and **E**, allow one to quantify over the branching structure of a transition system.

- **A**φ states that all the executions out of the current state satisfy property φ.
- **E**φ states that from the current state, there exists an execution satisfying φ.

The path quantifiers are mostly used in combination with the LTL operators, and they are easiest to understand in terms of the computation tree obtained by unfolding the Kripke structure. The **A** and **E** combinators on the one hand, **G** and **F** on the other hand, are often used in pairs as illustrated in Figure 1.8.

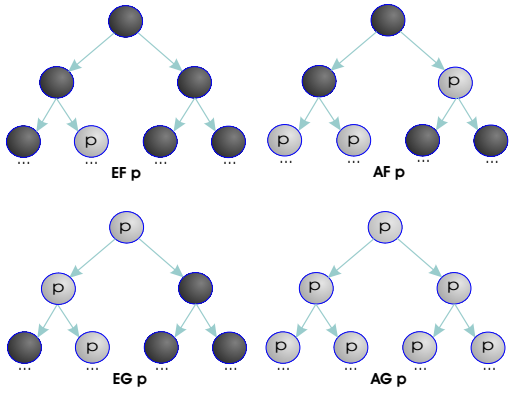


Figure 1.8: CTL Operators

Example 1.4. *Using temporal logic, the property "it is always possible to reach a state where tankB is ok or full" can be formalized as:*

$$AGEF(tankB = Ok \vee tankB = Full)$$

To understand this formula it is sufficient to know that $AGEF\varphi$ states that at any instant of any execution it would be possible to reach φ. Or, φ is always potentially reachable. This can be verified even if there exists an execution in which φ is never realized. Along every execution, the quantifier E allows expressing that alternative executions exist which would carry on the system behavior in different ways.

Tool Support

Some myth that sometimes is still being propagated is the following: *formal methods are not supported by tools* [20]. The concepts presented above take effect in *model checkers*; which are of course mechanized versions of the model checking technique. During the past few years, errors have been discovered, when using these model checkers, within larger and larger systems. The success and limitations of these tools lead to an important research activity in the field of verification. Although this research has a high theoretical

level, it systematically faces reality through the development of new tools, with increasing efficiency. We will mention a few of them.

To perform verification tasks on statecharts, we have decided to use the *Symbolic Model Verifier* (SMV) [87]. This tool has been developed by K. L. McMillan and has further evolved into Cadence SMV (CaSMV) [86]. Its input language provides features for describing finite state systems that range from synchronous to asynchronous. The proof assistant supports several compositional methods. These methods allow the verification of large, complex systems by reducing the verification problem to small problems that can be solved automatically by model checking. It provides a variety of such techniques: induction, circular compositional reasoning, temporal case splitting, and data type reduction.

SPIN [54, 55] was designed for simulation and verification of distributed algorithms. Its input language allows to describe the behavior of each process in the system as well as the interactions between them. Its key feature is the availability of several state space reduction methods: state compression, on-the-fly verification and hashing techniques. Properties to be verified can only be written in LTL.

Other model checkers are *DESIGN/CPN, UPPAAL, KRONOS*, and *HYTECH*. Interesting to note is that some *theorem provers*, like *PVS* [109] or *Isabelle* [100] provide model checking features. This approach allows state of the art model checkers to tackle intractably large or even infinite state spaces. Theorem proving has the advantage of effectively verifying infinite state models using axioms and proof rules but it has two main disadvantages. Firstly, it requires sufficient expertise in formal methods. Secondly, if and when a failed proof occurs, the theorem prover does not present a counterexample. As a consequence, only a small number of research groups has been able to apply this technique to industrial-scale software.

1.6.4 The Benefits of Model Checking

Model checking [26] is a formal method, a mathematical approach. We use it as part of the verification process. Why not using another formal method like for example theorem proving?

In [21] it is pointed out that most software developers still believe that formal methods are too hard, to tedious to be used in practice, that it requires effort, expertise and significant knowledge in order to be succesful. But this definitely does not hold for all types of formal methods. It might be true for theorem proving but it is false for model checking. Model checking is a fully automatic technique; once activated, for finite state possibly interleaving processes, it performs an exhaustive search to verify whether a logical formula

either holds or not. Consequently, this considerably reduces the amount of knowledge of the underlying formal model that is required to use this approach.

But there is another very straightforward reason. Statecharts are hierarchical state machines, i.e. finite-state machines whose states themselves can be other machines. They document the various modes ("states") that a class can go through, and the events that cause a state transition, together with the resulting actions. Since the design covers state machines, *model checking* [26] is the most appropriate technique to verify the design (or model) against the given specifications, as will be clarified later on.

Conclusion We have a formal method that is capable of independently verifying statechart diagrams. That formal method is part of a Java application. That application visualizes the counterexample returned by the model checker. Thus, software developers are not required to have any mathematical knowledge about the model checking technique at all. A single push on a button is enough to know whether their designs satisfies the requirements. *Admitted, do we, as software developers, still want more?*

1.6.5 The Benefits of Cadence SMV

Why have we chosen to use CaSMV and not for example SPIN [55]? As said before, the lack of understanding formal methods results in disasters, which is perhaps the main reason why they are still not trusted in industry [21]. The same can be said of tools supporting formal methods. The more difficult they are to learn and to use, the less they will play a role in the development life cycle.

To convince software developers of the applicability of formal methods and tools, we have chosen to integrate CaSMV in our application. It has a language that is easier and more natural to use comparing with other languages (e.g. Promela [55, 54], the language of the SPIN model checker). The language is based on a language for describing hierarchical finite concurrent state machines. As a consequence, the statechart diagrams are easily reflected in the CaSMV model. Moreover, a dedicated introductory course on using the model checker takes less than two weeks. The user gains success rather quickly; strengths and weaknesses of CaSMV are rapidly known.

More technically, we have chosen to use CaSMV for the following reasons. It is a tool for checking finite state systems against specifications in both temporal logics LTL and CTL. It also provides an input language for describing finite state systems that range from synchronous to asynchronous. Therefore

it is well-suited for reactive systems whose components are tightly connected through the exchange of (a)synchronous messages (events). Moreover, it is a proof assistant that supports several compositional methods. These methods allow the verification of large, complex, systems by reducing the verification problem to small problems that can be solved automatically by model checking. It provides a variety of such techniques including *refinement*, *circular compositional reasoning*, *induction*, *temporal case splitting* and *data type reduction*. For example, we have used the first two techniques to automatically decide whether a class implements its interface.

Conclusion We have integrated an easy to use tool that applies the model checking technique to finite state possibly interleaved concurrent models. Whenever a software developer wants to add or wants to change something in our automatic verification methodology, he only has to learn the main principles of model checking and, of course, he has to learn how to work with CaSMV. We hope that these two benefits will motivate software developers to integrate and to use formal methods more in the software life cycle. *Are you still hesitating?*

1.6.6 Some Succes Stories

Another motivation for the choice of model checking and CaSMV was also due to some success stories. Model checking was originally intended for the analysis of concurrent software systems, but nevertheless, the technique was first used in hardware verification. Let us enumerate some of the many success stories, whose list still continues to grow.

In the hardware community, model checking is a proven succesful verification technology. An impressive example that illustrates the power of both model checking and (Ca)SMV is the verification of the cache coherence protocol described in the IEEE Futurebus+ standard (IEEE Standard 896.1-1991). The purpose of [28, 81] was to use SMV to prove the correctness of the protocol design. However, with the model checker SMV, several bugs and potential errors were found. A similar success story: the model checker SMV was used to find a deadlock in a cache coherence protocol for a shared-memory multiprocessor [85]. Another similar success: using model checking the reason that data was lost on a high speed communications IC Chip could be identified [92].

Model checking has also proven to be effective in the software community. A number of properties of the Traffic Alert and Collision Avoidance System II (TCAS II) were successfully analyzed with the model checker SMV [25]. SMV was also used to analyze the A-7E aircraft software requirements [108];

several temporal properties were verified and/or falsified. Bandera [30], which is a toolset for extracting models from Java source, uses SMV (one of the supported model checkers) to apply model checking on these models.

1.6.7 The State Explosion Problem

Model checking is a method used to verify the behavioral correctness of several systems. In its basic form, it constructs a structure (e.g. infinite tree) that consists of all states that a system can reach and all transitions that a system can make between those states.

Model checking sounds like an almost ideal behavioral verification technique but it suffers from one big and fundamental problem: *state explosion* [26] and almost any system has a huge number of states. Often the size of a state space of a system tends to grow exponentially in the number of its processes and variables. The base of the exponentiation depends on the number of local states a process has, the number of values a variable may store, and so on.

With the advantage of model checking in mind, researchers have developed several methods in order to reduce the number of required states: abstraction, symmetry, induction, partial order reduction, and many others [26]. Abstraction, for example, is probably the most important technique for reducing the state space explosion problem. The technique is performed on a high-level description of the system, before the model of the system is constructed. Thus, abstraction avoids the construction of the unreduced model that might be too big to fit into memory. Some abstraction techniques reduce the size of the state transition graphs while others find a mapping between the actual data values in the system and a small set of abstract data values. On the other hand, partial order reduction is aimed at reducing the size of the state space that needs to be searched by model checking algorithms. To state it differently, the full state graph, which may be too big to fit in memory, is never constructed.

Slicing can be viewed as a projection, or as a flattening of a program. When used to slice models (like the previous examples), we'd better call it *model slicing* instead of *program slicing* to clarify the distinction between both techniques. The first one, constructs a slice based on a given correctness property while the latter one uses a collection of program statements to compute its slices.

Nevertheless, all reduction techniques have a common property. The reduced system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the reduced level. Moreover, the correctness of the verification is not affected i.e. the same results are

achieved when verifying properties at the actual system as at its abstracted version.

Program Slicing

Program slicing [112] is a program analysis technique that reduces programs to those statements that are relevant for a particular computation. A slice provides the answer to the question "What program statements potentially affect the value of variable v at statement s?" Mark Weiser [127, 128] introduced program slicing because he made the observation that programmers have some abstractions about the program in mind during debugging. When debugging a program one follows the dependencies from the erroneous statement s back to the influencing parts of the program. These statements may influence s either because they decide whether s is executed or because they define a variable that is used by s. Program slicing computes these dependencies automatically and thus assists the programmer in a lot of error prone tasks, such as debugging, program integration, software maintenance, testing, and software quality assurance.

Example 1.5. *Code Listing 1.1 shows a simple program that we want to slice. The slice for the statement* `print a`*, consists just of the black statements. The gray statements do not have an influence on the particular statement of interest, and therefore are not included into the slice.*

Code Listing 1.1 A Simple Program

```
read a
read b
read c
a = 2*b
print a
```

... and Model Checking

Program slicing is a technique for identifying portions of a program that can influence the way in which a collection of designated statements executes. For the purpose of model construction, we can use slicing to automate the process of eliminating portions of a program from a model without affecting the precision of the model. The idea is to extract a safe slicing criterion from the specification to be checked, e.g., temporal logic formulas. For example, this process may identify the set of statements that can directly influence the

truth or falsity of the propositions in the formula. Slicing on such a criteria leaves a resulting program that preserves all of the relevant information for the behaviors of interest as defined by the specification. This sliced program is used then as a starting point for the abstraction and specialization process.

Research has shown that it is a useful abtraction technique to tackle the state explosion problem when model checking is applied to software programs. Applying model checking to software requires that the program source code must be translated to a finite-state transition system that, of course, safely models the program behavior. Some researchers [30, 51] have developed a variation on program slicing to remove parts of the program's source code that are irrelevant for verifying a given correctness property.

Before model checking can begin, the model has to be adapted into a formalism accepted by a model checking tool. Yet another extension of program slicing has been succesfully used to slice models written in the input language of the model checker SPIN [93, 94].

Specifically slicing is used to reduce the statecharts of the RSML modeling language [52] in order to improve the understanding of the design specification. Another attempt to slice statecharts is developed in [43]. To continue, [125] presents a method to reduce the state space in model checking by slicing equivalent representations of UML statecharts with respect to a given correctness property. It is the latter methodology that we will optimize to achieve smaller models.

1.7 A Guided Tour Through This Thesis

This document reports both on things we have discovered and things we have built. It contains overviews on topics that are relevant for the context and the motivation of our work. How our work relates to the literature is always mentioned (mostly at the end of each chapter), and each chapter provides a specific functionality of our UML based verification approach (see the sections with title *Methodology Extended*).

Part I — Verification

Applying formal techniques in the development of embedded systems relies on a model of the system, a description of the requirements of the system and a technique to check that the requirements are satisfied by the model. In the first part, we propose a UML-based verification method to identify and remove errors, misconceptions, etc. during the design phase of embedded software development. In order to mechanise the technique that checks that

the conditions on the evolution of the system are met by the model, we propose model checking and the use of CaSMV (Section 1.6). This avoids that designers have to recover from faulty reasoning later on in the development process (Section 1.5). Remember that statechart diagrams are mainly used to model the evolution of embedded systems. And of course, the behavior of embedded systems can be captured by interacting finite state machines.

Chapter 2 — Semantics of UML Statecharts Figures 1.3-1.4 show that applying model checking to a UML design consists of several tasks. The first task is to convert the design into a formalism accepted by CaSMV. This modeling task is simply a compilation task that can be performed automatically but that is guided by several rules. Naturally, to guarantee a correct verification of system properties, the semantics of statecharts must be fully respected, otherwise verification does not make very much sense. In this chapter, a formal statechart semantics is constructed to ease the modeling task towards CaSMV. To do that, statecharts are first transformed into an equivalent format, called Extended Hierarchical Automata [68, 72, 124], and the formal semantics of statecharts are written in terms of the latter ones.

Contribution: We present the content of this chapter as highly relevant background material. The only contribution is that I have written a formal semantics (that differs from that used in [36, 73]) in order to have a clear connection between the formalism accepted by CaSMV [86].

Chapter 3 — The Model of a Standalone Statechart As mentioned, this thesis especially concentrates on the behavioral part of UML, namely the statechart diagrams, because they are the most complex formalism used in UML (therefore errors occur most likely here) and have some specific features, like hierarchy and concurrency, which require non-trivial verification methods. Our examination is focused on embedded (control) systems since statecharts allow to construct a state-based model of such systems and the reaction to external events. In this chapter, it is assumed that the behavior of the embedded system is specified by a standalone statechart. To carry out the verification automatically using CaSMV, this chapter allows us to translate each standalone statechart into a CaSMV model, of course, using the formal semantics defined in Chapter 2.

Contribution: The methodology to verify the behavior of a system, specified within a standalone statechart, is not unknown in the literature. All the methodologies have one thing in common: they translate the hierarchical structure of the statechart model to the input language of a particular model

checker. So, the black box of the different approaches will be more or less the same; only the details differ. Positively stated, our methodology tries to retain the quite complex behavioral semantics of UML statecharts, while the existing approaches make some significant simplifications. This has the advantage that our verification methodology can handle more realistic behavioral designs. For published material, see the next chapter.

Chapter 4 — The Model of Communicating Statecharts A behavioral model described as a statechart is based on modes and transitions, on events, on conditions and on different types of data items too. But a realistic behavioral design of a system is not described as a single statechart. There are many examples where the whole system is decomposed into several statecharts, but only some of them are active at any given time. For example, there may be a factory with several independent machines (chip placement machines, screening machines, etc.) that make up an assembly line. Each piece of equipment may be controlled by a statechart that describes its behavior. Not suprisingly, for such complex systems, verification (using the same model checker CaSMV) is definitely needed to find incorrect behavior.

Contribution: In this chapter, the formal semantics defined in Chapter 2 is extended into a behavioral semantics for communicating statecharts. As before, this semantics is used to generate a corresponding CaSMV model. This bridges a big gap in the literature, and enables developers to verify designs of quite complex systems. Therefore this chapter can be seen as an important addition to contemporary work. The technical content of this paper has been summarized in a one-page abstract and a poster of the "Proceedings of the 32nd Spring School in Theoretical Computer Science" [116], appeared as part of a paper in the proceedings of the "Proceedings of the Dutch Proof Tools Day" [115], and was visualized in a second poster, which was presented at the "PhD FWET Symposium" [119].

Chapter 5 — Specification Correctness Towards verification purposes, it is necessary to state the properties that the design must satisfy. System properties are usually given in some temporal logical formalism to assert how the behavior of the system evolves over time. How to write properties and how to translate them in the language of CaSMV is outside the scope of this thesis. However, what we will discuss are some problems with temporal formulas that arise after the model transformation has been performed in the first task.

Contribution: The technical content of this chapter appeared as part of a one-

page abstract and a poster of the "Proceedings of the 32nd Spring School in Theoretical Computer Science" [116], and appeared as part of a paper in the proceedings of the "Proceedings of the Dutch Proof Tools Day" [115].

Chapter 6 — Protocol Conformance If you are familiar with object-oriented methods, you will be aware of the concept of a class and an interface. The UML allows that classes use *behavioral state machines* — also called statecharts — to describe their piece of system functionality as a sequence of events an object reacts to, together with the resulting behavior. The UML also allows that interfaces use *protocol state machines* to focus only on allowable sequences of behavior invocations on a class, but, without having to show its behavior. Having two views — a behavioral and a protocol view — for components of a system gives rise to the problem of model consistency: both statecharts have to be consistent and not contradictory; a class needs to implement its interface. Otherwise, an implementation of the designed models will not be feasible, thereby making them useless.

Contribution: This chapter provides a methodology to tackle this consistency problem using useful verification features of CaSMV. A one page abstract appeared in the "Proceedings of the Joint BeNeLuxFra Conference in Mathematics" [120]. A condensed version of this chapter is to appear in the "Bulletin of the Belgian Mathematical Society" [121].

Part II — Optimization

As mentioned enough already, model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. CaSMV is a formal verification tool, which is quite effective in automatically verifying properties of combinational logic and interacting finite state machines. The major problem of model checking is that the state spaces arising from practical problems are often hugh, generally making exhaustive exploration infeasible. Dependency analysis and a corresponding slicing algorithm is one way to improve scalability. This is the topic addressed in the second part of my thesis.

Chapter 7 — The Basics of Slicing Hierarchical Automata This chapter presents a method for slicing hierarchical automata with respect to the properties to be verified. The algorithm can remove the hierarchies and the concurrent states which are irrelevant to the property. This results

in smaller hierarchical automata. As a consequence the state space during model checking is efficiently reduced.

Contribution: The technical content of this chapter appeared in [125], as joint work of Wang Ji, Dong Wei, and Qi Zhi-Chang.

Chapter 8 — Internal Broadcasting: As Rich As Needed Concurrent programs have additional dependencies called *interference dependencies*. A node S_1 is interferent dependent on node S_2 if S_2 defines a variable v, S_1 uses variable v, and S_1 and S_2 execute in parallel. A simple slicing algorithm that does not carefully threat such inference dependencies will produce imprecise slices, i.e. a smaller slice could be returned. The same is true when slicing hierarchical automata.

Contribution: The algorithm presented in Chapter 7 presents a naive slicing algorithm, in the sense that interference dependencies are not carefully threated. The goal of this chapter is showing how the algorithm presented in Chapter 7 can be improved to yield an algorithm that is efficient yet effective for reducing the number of interference dependencies used in slicing statecharts with concurrent states. This way, far more precise slices can be returned by the algorithm. The technical content of this chapter was first presented on the "TCS Seminar in Amsterdam" [118]. It was a second time presented on the "FNRS Contact Day in Liège" [117].

Part III — Illustrations

The title of the last part could also have been: "The Theory into Practice". Chapter 9 illustrates the verification procedures: the statecharts, the CaSMV models, the interpretation of the counterexamples, etc. Chapter 10 extensively shows how the slicing algorithm works. Finally, Chapter 11 uses examples to explain that carefully considering interference dependencies may yield smaller slices.

Part I

Verification

CHAPTER 2

Semantics of UML Statecharts

The eye sees only
what the mind is prepared to comprehend.
Henri Bergson.

Almost all embedded systems are event-driven, meaning that they continuously wait for the occurrence of some external or internal event (e.g. an arrival of a data packet). After recognizing the event, such systems react by performing the appropriate computation that may include generating events that trigger other internal software components. Once the event handling is complete, the software goes back to a dormant state (e.g. an idle task) in anticipation of the next event. This reactive system behavior is often described by graphical state based notations. Among others, UML statecharts [38, 96, 105] is one example for these notations and they have found widespread acceptance in the industry.

This thesis is about verifying UML statecharts based on model checking. So what is a statechart precisely? That is one of the topics of this chapter. Additionally, to ease verification, statecharts are first transformed into Extended Hierarchical Automata [46, 68, 72, 124].

The formal verification of a behavioral design involves proving the statechart's consistency with its specification. To do this, the meaning of these statechart diagrams needs to be defined and understood. Therefore, it is necessary to provide a precise formal semantics for constructs of the behavioral design. This is another topic of this chapter. We will provide such a semantics for the hierarchical representation of a single statechart. It's presented in such a way that the connection between the formalism, accepted by CaSMV [86], becomes clear later on. Moreover, the semantics will serve as the basis for the semantics of communicating statecharts.

2.1 Finite State Machines

A program that sequences a series of actions, or handles inputs differently depending on what mode it's in, is often implemented as a *Finite State Machine* (FSM) [7, 79]. Such machines (Definition 2.1) are an efficient way to specify constraints of the overall behavior of a system. Being in a state means that the system responds only to a subset of all allowed inputs, produces only a subset of possible responses, and changes state directly to only a subset of all possible states.

Definition 2.1 (Finite State Machine (FSM)). *A finite state machine is an ordered four tuple* $\mathcal{M} =< Q, \Sigma, \delta, q_0 >$ *where* Q *is a finite set of states,* Σ *is the input alphabet — a finite set of symbols,* q_0 *is the initial state — a member of* Q*, the transition function* δ *maps the states and inputs onto a set of state transitions.*

The behavior of the machine is more easily understood when represented graphically in the form of a *state transition diagram.* In each particular state of the machine there can be zero or more transition rules that are executable. If precisely one transition rule is executable, the machine makes a *deterministic* move to a new control state. If more than one transition rule is executable a *nondeterministic* choice is made to select a transition.

Applied to reactive system objects, a FSM specifies the events of interest to a reactive object, the set of states that object may assume, and the actions (and their order of execution) in response to incoming events in any given state. This is crucial in many systems because the allowable sequences of primitive behaviors may be restricted.

Example 2.1. *A 10-bit counter [41], counting on event a and issuing overflow after 1024 occurences, can be represented as a FSM, as illustrated in Figure 2.1.*

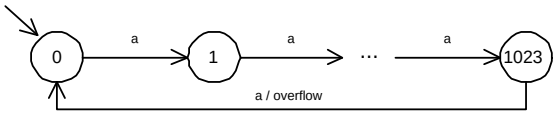


Figure 2.1: A 10-bit Counter as a Finite State Machine

2.2 UML Statecharts

Clearly, Example 2.1 shows the most important shortcoming of FSMs; the total number of states and transitions increase exponentially as the system complexity increases. The negative side effect of this is that the FSM becomes an unstructured and chaotic state diagram. To solve this problem David Harel proposed statecharts [49] which are extensions of FSMs. They form the basis for UML statecharts [96] which extend the properties of Harel's automata with some additional features and some modifications in the semantics.

Example 2.2. *The statechart of the 10-bit counter [41] (Example 2.1) is shown in Figure 2.2. Clearly, syntactic sugar is added to finite state transition diagrams such that suitable abbreviations for unwieldly diagrams become available. A transition is labeled with an event (e.g. the event* `a`*), a guard (e.g.* `[x<1023]`*), and an action (e.g.* `x:=x+1`*). The transition, labeled with* `a[x<1023]/x:=x+1`*, is taken when the event* `a` *has occurred and when the guard is satisfied (i.e. the event has not yet occurred 1024 times). While being taken, the action* `x:=x+1` *is executed. Obviously, the variable* `x` *holds the number of occurrences of the event* `a`*.*

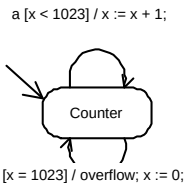


Figure 2.2: A 10-bit Counter as a Statechart

Objects have both behavior and state or, in other words, they do things and they know things. Some objects do and know more things, or at least more complicated things, than other objects. Some objects are incredibly complicated, so complex that developers can have difficulty understanding them. To understand complex classes better, particularly embedded classes that act in different manners depending on their state, one or more UML state machine diagrams should be developed.

UML statecharts [38, 96, 105] are hierarchical automata associated to objects (class instances) to model their behavior. A statechart is a complete graphical characterization of all the desired behaviors of an object during in its lifecycle. To be more precise, statecharts, consisting of states and

transitions, convey how objects behave through time as a result of the objects reactions to events from the rest of the universe.

2.2.1 States

A *state* depicts a situation where the object satisfies some condition, performs some activity, or waits for some event. Reactions include changing states, executing internal actions and sending events to possibly other objects. The event-driven nature of statecharts is especially useful in modeling reactive systems. Thus, each state represents a distinct context for the behavior of the system. States have a *status*, meaning that they are either *active* or *inactive*. When a state is active, the system is said to be in that state.

Example 2.3. *A state captures the relevant aspects of the system's history very efficiently. For example, you can say that a computer keyboard is either in the shifted state, or the default state. The behavior of a keyboard (Figure 2.3) depends only on certain aspects of event history, namely whether the Shift key has been depressed, but not, for example, on how many and which specific characters have been typed previously [102] . Note that the behavior only captures upper case and lower case letters.*

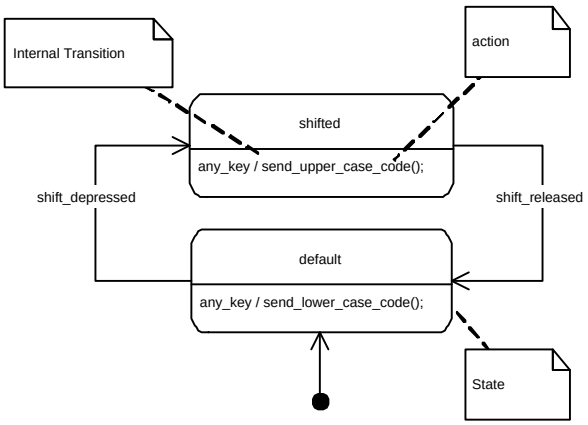


Figure 2.3: Statechart Diagram of a Keyboard

Each state possibly contains multiple *internal actions*. An action is best described as a task that takes place within a state e.g. pieces of code to implement behavior in that context. Table 2.1 gives a brief overview of different internal actions.

Action	Description
Entry Actions	Entry actions are executed at the moment the state is entered. Such actions are often used to perform setup needed within a state.
Internal Transitions	These transitions have a source state but no target state. They represent reactivity within a state, and when fired, the active state does not change as a result of it. Internal transitions can be useful for modeling interrupt actions that do not change the state e.g. putting up a help screen.
Exit Actions	Actions which are performed whenever the state is exited. If a transition leaves a state, its exit action is executed before the action on the transition and the entry action of the new state.

Table 2.1: Internal Actions of a State

States may be *hierarchical* in the sense that it can have distinct subcontexts represented as substates. The hierarchical state that contains other states is called the parent state of the contained children (sub-) states. A child state cannot be active if the parent is inactive. As parent states change from an inactive state to an active state, at least one child state necessarily becomes active.

The UML provides two different types of hierarchical states often called *composite states*. An *orthogonal composite state* (an AND state) is composed of several concurrent regions (= sequential composite states, the OR states) graphically separated by dashed lines. If an AND state is active, all its regions are active. Each OR state is composed of AND/OR states. If a *sequential composite state* is active, only one of its substates is active. A *simple state* is a state not composed of any substates. The *root state* is a special hierarchical state, since it is present in each statechart, and it is always active. This state is the outermost composite state of the statechart diagram, but is not drawn always explicitly.

Example 2.4. *Consider a requirement that states that a user should not have to wait for his/her browser to be able to work with his/her email application [106]. In considering this requirement, it would be necessary to specify these two activities as concurrent, as shown in Figure 2.4 (especially in considering the time that it takes to launch your average web browser and email application). Whenever the automaton is in state UserConnected both regions (Email and Internet) are considered active. The states of these*

regions are exclusive; the system can never be in both their states at the same time.

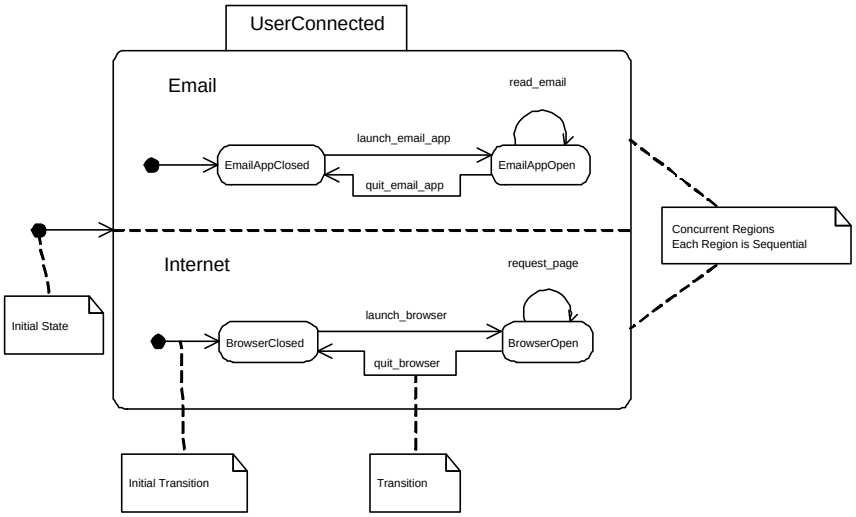


Figure 2.4: Statechart Diagram of a User Connection

In addition to the above primitive constructs, *pseudostates* such as initial, final and history states are used to extend the notation. An *initial state* is the source of a transition which points to the default substate of the composite state whereas a *final state* is one in which no transitions lead out of. Default substates are the ones that an object is in when it is first created. A *history state* records the most recent active state information of its containing state. History states are useful for when the device has left a hierarchical state, done something else for a while, and now needs to get back to the hierarchical state and context. There are also other pseudostates (join, fork, conditionals, etc.) used to connect multiple transitions into more complex state transitions paths (see Section 2.2.2).

2.2.2 Transitions

The job of transitions is to specify when and to which states the object can switch. Transitions represent potential pathways among the states of the object. The progression of one state to another is triggered by an event that is either internal or external to the object. The syntax for transitions is:

event-trigger (parameters) [guard]/action-list

Table 2.2 explains the fields of transitions in greater detail. All of these fields are optional. Even the event-trigger may be omitted in the case of an *null-triggered transition* to be taken when a state completes its activities, if any, or immediately upon entering the state, if not.

Field	**Description**
Event-trigger	The name of the event triggering the transition.
Parameters	Events may have parameters (comma-separated list) and these parameters may be used by the actions associated with the transition.
Guard	A boolean expression that is evaluated when the event occurs. The transition is taken only if the guard evaluates to TRUE.
Action-list	A comma-seperated list of operations executed as a result of the transition being taken.

Table 2.2: Transition Syntax

A *simple transition* indicates that the system may change its state and perform a sequence of actions when a specified event occurs and a specified guard condition is satisfied. Such transitions represent directed relationships between a source state vertex and a target state vertex.

On the other hand, UML provides also *compound transitions*. Such transitions represent paths made of one or more transitions which are linked by pseudostates, originating from a set of states and targeting a set of states. A compound transition is enabled when all the source states are occupied, a specified event occurs and the guard is satisfied. After a compound transition fires, all of its destination states are occupied. Otherwise stated, compound transitions consists of multiple segments.

To show *parallel behavior*, *fork segments* or *join segments* are used. The fork (top of Figure 2.5) has one transition entering and any number of transitions exiting, all of which will be taken. The exiting transitions are connected to concurrent regions and represent splitting of control. The join (bottom of Figure 2.5) represents the end of the parallel behavior and has any number of transitions (originate from concurrent regions) entering, and only one leaving.

To show *conditional behavior* a branch or a merge is used. The top diamond of Figure 2.6 is a *branch* and has only one transition flowing into it and any number of mutually exclusive transitions flowing out. That is, the guards on the outgoing transitions must resolve themselves so that only one is followed. The bottom diamond of Figure 2.6 is a *merge* that is used to end

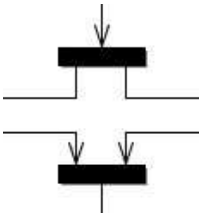


Figure 2.5: A Fork and a Join Segment

the conditional behavior. There can be any number of incoming, and only one outgoing transition.

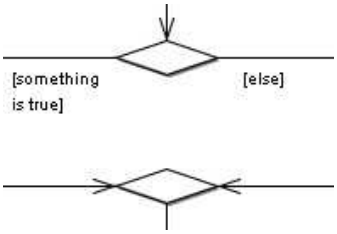


Figure 2.6: A Branch and a Merge Segment

UML features also *interlevel transitions*. Such transitions cross state boundaries.

2.3 Informal Operational Semantics

Semantics of UML is a means to understand how UML should be used, and to ensure that when UML models are communicated there is a common shared understanding of what they mean. Therefore, if we ever want to be able to verify behavioral properties of statecharts, we must have a clear understanding of its semantics. A *semantics definition* is thus a necessary prerequisite. The term *semantics*, as used in this thesis, refers to the run-time interpretations of UML statecharts.

For some, the only acceptable definitions of language semantics are the ones that are expressed in some well-known mathematical formalism. Despite, UML found this not the most suitable approach [14], although their intent is to define the semantics of state machines very precisely. This is because UML is intended to model complete systems across a broad spectrum of different application domains [61]. Therefore, there are a number of

variation points to allow for different semantic interpretations that might be required in several domains.

iLogix Rhapsody [58] is a CASE - Tool for embedded systems software development. Rhapsody is claimed to be "The Industry's first and only UML based object-oriented analysis, design and implementation tool for embedded systems and software developers." That's the reason why we have used it during the last four years. Rhapsody offers automatic code generation with additional simulation functionality. Many of the fundamentals of the statechart's semantics, as implemented in the current version of the tool, have been adopted in the UML. The main difference between both semantics is that, as mentioned previously, UML leaves the door open for many semantic variation points.

The operational semantics followed in this thesis, is the semantics offered by Rhapsody [48].

2.3.1 Active Configuration

Obviously, embedded systems are composed of many classes. Some of these classes are reactive and thus have an associated statechart describing its behavior. Others are data-driven and do not necessarily have an associated statechart. During run-time possibly many objects exists and each can be in a different active configuration. This is because for each new instance of a reactive class, a new statechart is born.

An *active configuration* is defined as a maximal collection of active states i.e. as the maximal set of states the system resides in simultaneously, including the root state. A state becomes active when it is entered as a result of some transition, and becomes inactive if it is exited as a result of a transition. If a sequential composite state is entered, exactly one of its substates is added to the configuration. If a concurrent composite state is entered, all of its substates are added to the active configuration.

Example 2.5. *Returning to Figure 2.4, a possible active configuration can be* $\{root, UserConnected, EmailAppClosed, BrowserClosed\}$ *or* $\{root, UserConnected, EmailAppOpen, BrowserOpen\}$.

2.3.2 Event Handling

The behavior of an embedded software system can be thought of as a black box that continuously receives some input events and reacts by producing some output events. This output may in turn affect the production of later

input events by the environment. To handle such events, UML provides the following extremely important key components:

- An *event queue* that holds incoming event instances until they are dispatched.
- An *event dispatcher mechanism* that selects and dequeues event instances from the event queue for processing.
- An *event processor* that processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question. This component is simply referred to as the "state machine".

2.3.3 Stable State Configuration

An active configuration is a *stable state configuration* of an object if all actions (entry/exit/internal activities) are completed and if no further transitions are possible without dispatching an event i.e. it is impossible to enable and fire null-triggered transitions.

Example 2.6. *Let's return to the statechart diagram of Figure 2.4 and suppose that configuration* $\{root, UserConnected, EmailAppClosed, BrowserClosed\}$ *is active. This configuration would be stable as well since an event (e.g. launch_browser, launch_email_app) must be dispatched to reach either* $\{root, UserConnected, EmailAppClosed, BrowserOpen\}$ *or* $\{root, UserConnected, EmailAppOpen, BrowserClosed\}$*. It would be instable if it had an outgoing transition that was not labeled by an event.*

In the context of a single statechart, the start chart may react on different kinds of events. Each statechart uses so-called *interacting events* to communicate with the outside world e.g. to interact with a user. Besides interacting events, a statechart may generate events to represent the completion of internally generated actions. These events are used by the statechart to support the procedural flow of control. From now on, we refer to them as *flow events*.

2.3.4 Step Semantics or Run-to-Completion

The behavior of a system is described as a set of possible runs. A *run-to-completion* (RTC) step brings the state machine from one stable state configuration to another stable one. Therefore, a *run* consists of a series of detailed stable snapshots of the system's situation. The first one in the

sequence is the initial configuration and each subsequent configuration is obtained from its predecessor by executing a step.

In fact, a RTC step is the period of time in which events are accepted and acted upon. At the beginning of a step, an event is firstly dispatched from the queue and then processed. Intuitively, this means that once an event has been dispatched, some transitions are enabled to fire, and thereafter, the system evolves on its own until no more transitions can be taken. Then, the dispatcher has to be called once again. Events are assumed to never occur exactly at the same time. More precisely, if two events occur at the exact same time, they can be processed as if they had occurred in either order, with no loss of generality.

What does it mean that the system evolves on its own? After reacting to a message, the statechart may reach a configuration in which some of the active states have outgoing null-triggered transitions that can be enabled for firing. In this case, the system evolves on its own, meaning that these null-triggered transitions will be taken until a stable state configuration is reached. Once a stable state configuration is reached, the reaction to the event is fully completed and control returns to the dispatcher. At this point, the event is entirely consumed and is therefore no longer available for processing. The next event can be dispatched and processed now. Events that cannot be served immediately to a stable configuration are discarded meaning that the current stable configuration does not evolve to another stable one; the event does not enable any transitions.

Example 2.7. *Looking at Figure 2.4, the RTC step between configurations* $\{root, UserConnected, EmailAppClosed, BrowserClosed\}$ *and* $\{root, UserConnected, EmailAppClosed, BrowserOpen\}$ *consists in the execution of a single transition, which is labeled with the event-trigger* $launch\_browser$*. Here, the evolution to the second configuration happens without going through intermediate snaphots of the system. This is quite clear because the statechart does not have null-triggered transitions i.e. in each configuration, an event must be dispatched and processed to reach a subsequent one.*

The execution of a step does not necessarily take zero time. The time a step will take depends on the actions that are performed while taking the step. Therefore, the RTC step is assumed to be *ininterruptable* i.e. higher-priority events cannot interrupt the handling of other events thereby completely avoiding the internal concurrency issue. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation; instead the event will be placed in the event queue when it is received during the execution of a step. Obviously, the assumption simplifies the transition function of the state machine since incoming events

are processed only after the state machine has reached a well-defined stable state configuration. The practical meaning of these semantics is *thread protection*, allowing the state machine to safely complete its RTC step without concerning about being interrupted in mid-transition by a subsequent event.

Summarized, each RTC step is composed of *microsteps* between two stable configurations, as illustrated in Figure 2.7 [48]. It asserts that events are consumed one by one, where the processing of the next event does not start before the previous one has been fully consumed. As a response to an event, the system, being in a stable configuration, undergoes a series of microsteps (reflected by the injection of several transitions) until it reaches another stable configuration. At that point, the system is ready to react to the next event.

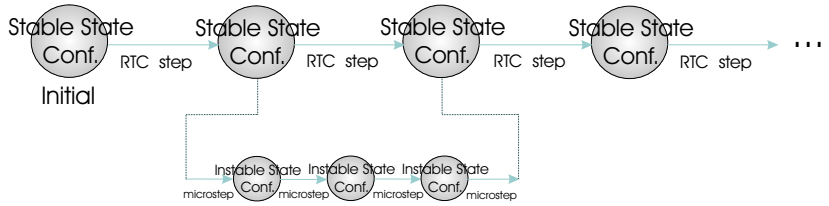


Figure 2.7: The Run-to-Completion Step

The Dispatcher and the Event Queue

The role of the *dispatcher* is to find an event that can be accepted in the current configuration of the object. In general, more than one event can be available in the environment, and therefore they are stored in a *First in First Out* (FIFO) *queue*. This queue not only stores the events but also forms the basis for the dispatching operation since the dispatcher works on this queue. When trying to dispatch an event, the dispatcher considers the events in the queue in the order they were put in.

The processing of the dispatched event consists firstly of selecting a maximal set of *non-conflicting transitions* among the enabled ones. The selected transitions are then fired, and as a consequence there is an implied action sequence. Firstly, the source states are exited and their exit actions are executed, where deeper states are exited before their parent states. Secondly, the actions attached to the transitions are executed. The closer the action is to the source state, the earlier it is evaluated. Finally, the target states are entered and their entry actions are executed. Parent states are entered

before substates. Note that in case of orthogonal regions, some orders are undetermined. Conceptually, actions are instantaneous, atomic and non-interruptible.

What is the meaning of conflicting transitions? In UML, it is possible for more than one transition to fire at the same time. When this happens, such transitions may be in conflict with each other. Take the case of two transitions originating from the same state, triggered by the same event, but with different guards. If both guard conditions are true, there is a conflict between the two outgoing transitions. Only one of them must be fired. But which one and what criteria can be used to make the choice? In such a situation, the selection of which transition will fire is based on a priority scheme. A transition has a higher priority if its source state is a substate of the source of the other one. If the conflict cannot be resolved using priorities, UML allows that they may be fired non-deterministically.

Example 2.8. *Figure 2.8 defines a well-formed statechart diagram (adapted from [60]). There is a concurrent composite state A with regions B and C. Transition t_1 has a lower priority than transition t_3. Transition t_2 has lower priority than transitions t_4 and t_5. The conflict between the latter ones cannot be resolved using priorities because their source states are equal. As a consequence, they will be fired non-deterministically. Transitions t_3, t_4 and t_5 are some of the interlevel transitions and transition t_1 is concurrent with transitions t_2, $t_4$and t_5.*

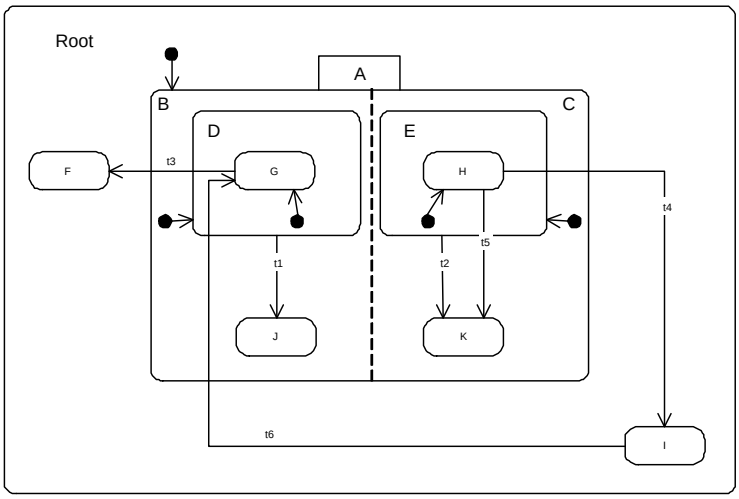


Figure 2.8: A Sample statechart to Illustrate Transition Priorities

What happens in the case concurrent states are integrated in the state machine? In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event; as many as one transition in each region in the current state configuration. The priority scheme is used here again to solve possible conflicts among transitions, of course, in each region separately. The order in which selected transitions fire is not defined. Each orthogonal region in the active state configuration that is not decomposed into orthogonal regions can fire at most one transition as a result of the current event. When all orthogonal regions have finished executing the transition, the current event is fully consumed, and the run-to-completion step is completed. Thus, a RTC step applies to the entire state machine. However, it is possible to define state machine semantics by allowing the RTC steps to be applied concurrently to the orthogonal substates of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement.

2.4 Extended Hierarchical Automaton

Extended Hierarchical Automata (EHAs), which form the structural basis of our formal operational semantics, were introduced in [46, 68, 72, 124] to provide an alternative equivalent representation and a formal operational semantics for statechart diagrams by a small number of complex transition rules. EHAs can be considered as a formal syntax of UML statecharts, describing the statechart elements in a concise format, resolving the problem of inter-level and composite transitions by using special labels. An EHA (Definition 2.3) is built as a parallel and/or hierarchical composition of sequential automata (Definition 2.2) whose states themselves can be other automata. We follow the definition from [124], which is a variant of the one introduced in [68].

Every sequential automaton $A \in F$ characterizes an extended hierarchical automaton on its own: intuitively, such an extended hierarchical automaton is composed by all those sequential automata which lay below A, including A itself, and has a refinement function ρ_A which is a proper restriction of ρ. An EHA can be regarded as a kind of abstract faithful syntax of UML statecharts, whose syntax is abstracted and the essential parts are reserved and presented structurally.

Definition 2.2 (Sequential Automaton). *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of states, s^0_A is the initial*

state, λ_A is a finite set of labels, and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the transition relation.

Definition 2.3 (Extended Hierarchical Automaton). *An EHA $H = (F, E, \rho, A_0, V)$ is a 5-tuple, where F is the set of sequential automata with mutually disjoint sets of states, E is a finite set of events, and V is the set of variables. $\rho : \cup_{A \in F} \sigma_A \rightarrow 2^F$ is a refine function, which defines a tree that satisfies:*

- *there exists a unique root automaton $A_0 \in F$ having no parent states: $\nexists s \in \cup_{A \in F} \sigma_A$: $A_0 \in \rho(s)$;*
- *every non-root automaton has exactly one ancestor state: $\forall A \in F \setminus \{A_0\}$, $\exists_1 s \in \cup_{A' \in F\ \{A\}} \sigma_{A'}, A \in \rho(s)$;*
- *there are no cycles: $\forall s \notin \rho^*(s)$.*

A state s for which $\rho(s) = \emptyset$, $\mid \rho(s) \mid = 1$, $\mid \rho(s) \mid \geq 2$ holds is said to be a simple state, a sequential composite state, and a concurrent composite state respectively.

Example 2.9. *The extended hierarchical automaton of our sample statechart (Figure 2.8) is given in Figure 2.9 and the connection between both representations is quite clear. The refinement function is given by the dotted gray arrows. Default states are represented by thicker rounded rectangles. We have: $F = \{Root, A, B, C, Ds, Es\}$; $A_0 = Root$; $\rho(A) = \{B, C\}$; $\rho(D) = \{Ds\}$; $\rho(E) = \{Es\}$; all the other states are basic.*

The set of events is the union of several sets i.e. $E = E_i \cup E_f \cup \varepsilon$. Each subset denotes the set of *interaction events*, the set of *flow events* and the *null-event* respectively. An interaction event is an event generated by the end-user of the system e.g. the event *inter_poweron* is generated at the moment the end-user turns a machine on. A flow event is internally generated by the system, so that the system can control its own execution. The null-event is used to characterize null-triggered transitions.

The claim that extended hierarchical automata are alternate equivalent representations of statecharts is obtained only when transition labels of transitions t of sequential automata $A \in F$ are required to be 5-tuples (sr, ev, g, ac, td) where

- the *source restriction sr* refers to the source(s) of the transition in the statechart i.e. it captures the "real" source(s) of the transition;
- *ev* is the event which triggers the transition;

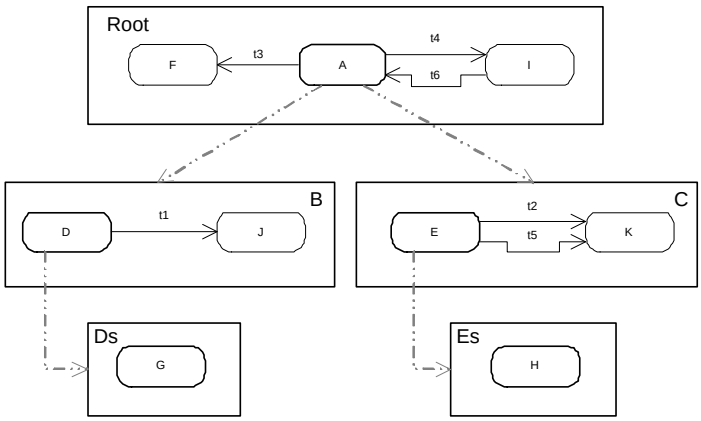


Figure 2.9: A Sample Extended Hierarchical Automaton

- g is the *guard*;
- ac is the *sequence of actions* to be executed;
- the *target determinator* td is the target(s) the transition enters in the statechart.

The source restriction and the target determinator play a major role in the representation of compound/interlevel transitions[1]. We use the following functions SR, EV, G, AC, TD defined in the obvious way; for a transition $t = (s, (sr, ev, g, ac, td), s')$ of a sequential automaton $A \in F$ ($\mathcal{S}(A)$ and $\mathcal{T}(A)$ are defined in Definition 2.4):

Function	Returns	Practical
SRC	source state in the EHA	$SRC(t) = s(\in \sigma_A)$
SR	source(s) state in the statechart	$SR(t) = sr(\in \mathcal{S}(A))$
EV	trigger event	$EV(t) = ev(\in E)$
G	guard	$G(t) = g$
AC	action list	$AC(t) = ac$
TD	target(s) state in the statechart	$TD(t) = td(\in \mathcal{T}(A))$
TGT	target state in the EHA	$TGT(t) = s'(\in \delta_A)$

Table 2.3: Functions Related to Transition Labels

An illustration of these functions is given in the following example:

[1]The semantics of the source restriction and the target determinator slightly differs from the one defined in [46, 72].

Example 2.10. *Table 2.4 illustrates the transition labels after transforming Figure 2.8 into Figure 2.9. You see that $SRC(t3)$ is the source of $t3$ in the EHA while $SR(t3)$ is the source of $t3$ in the statechart. A similar explanation for $TGT(t6)$ and $TD(t6)$. Note that the source restriction is always filled in whereas the target determinator is not always given.*

t	t1	t2	t3	t4	t5	t6
SRC(t)	$\{D\}$	$\{E\}$	$\{A\}$	$\{A\}$	$\{E\}$	$\{I\}$
SR(t)	$\{D\}$	$\{E\}$	$\{G\}$	$\{H\}$	$\{H\}$	$\{I\}$
EV(t)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
G(t)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
AC(t)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
TD(t)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{G\}$
TGT(t)	$\{J\}$	$\{K\}$	$\{F\}$	$\{I\}$	$\{K\}$	$\{A\}$

Table 2.4: Transitions Labels as Used in the Sample EHA

2.5 Formal Operational Semantics

The first step towards model checking UML statecharts is to provide a corresponding formal operational semantics. The semantics is defined over EHAs and is based on Kripke structures. Our notation is (slightly) different from that used in [36, 73] in such a way to have a connection between the formalism accepted by CaSMV [86].

Definition 2.4. *For $A \in F$, the automata, states, and transitions under A are defined respectively as*

$$\mathcal{A}(A) = \{A\} \cup (\bigcup_{A' \in \rho(\sigma_A)} \mathcal{A}(A'))$$

$$\mathcal{S}(A) = \bigcup_{A' \in \mathcal{A}(A)} \sigma_{A'}$$

$$\mathcal{T}(A) = \bigcup_{A' \in \mathcal{A}(A)} \delta_{A'}$$

2.5.1 Configuration

A configuration denotes a global state of an extended hierarchical automaton, either active or inactive. A global state is composed of local states of component sequential automata. This means that the global state of a configuration is derived from the local states of the set of sequential automata in a top-down manner starting from the root automaton.

Let $H = (F, E, \rho, A_0, V)$ be an EHA. We formally define a configuration as follows:

Definition 2.5 (Configuration). *A configuration of H is a tuple $\mathcal{C} = \langle \zeta, v, q_e, hist \rangle$ with*

- *$\zeta \subseteq \mathcal{S}(H)$ such that (i) $\exists_1 s \in \sigma_{A_0} : s \in \zeta$ and (ii) $\forall s, A : s \in \zeta \wedge A \in \rho(s) \Rightarrow \exists_1 s' \in A : s' \in \zeta$;*
- *v is the variable valuation;*
- *q_e is the event queue i.e. a FIFO queue containing events of $E \setminus \varepsilon$ and*
- *$hist$ is the history mapping that associates to each automaton the last visited substate i.e. $A \in F \rightarrow \sigma_A \cup \{nil\}$.*

For $A \in F$ the set of all configurations of A is denoted by $\mathcal{C}_A$.

In constrast to Section 2.3, the semantics of a configuration has changed since additional information is added to it. Now, not only states belong to configurations but also the event queue is part of it. This cannot be a surprise because the event queue is used thoroughly while the state machine is executing. Either an event is dispatched, to activate an RTC step, or new events are generated, as the result of some actions during an RTC step. The same reasoning can be applied for the other elements belonging to the configuration.

2.5.2 Priority

The UML approach to transition selection deals with the sets of exit states w.r.t. transitions. These sets include all states left by a transition in a step. In addition two transitions t and t' are defined to be in *conflict* if there is a common state in the exit state sets. Thus, there is at least one state that they both exit.

Priorities resolve some, but not all, transition conflicts. State hierarchies are used to define priority between conflicting transitions. A transition t' has higher priority than a transition t if one of its sources is located in a substate of a source of t. Thus, the state priority allows the transition to be taken if

there is no higher priority active state. If the priority cannot be solved then both transitions fire non-deterministically. As a direct consequence, UML's priority scheme is based on state levels and source states for which an order relation can be defined. Formally defining a priority rules scheme involves defining an order relation on the states of the automaton.

Definition 2.6 (State Precedence). *The state precedence for $s, s' \in \mathcal{S}(H)$, is defined as follows:*

$$s \prec s' \Leftrightarrow s' \in \mathcal{S}(\rho(s))$$

$\preceq$ denotes the reflexive closure of $\prec$. Relation $\preceq$ is a partial order relation.

Example 2.11. *Referring to Figure 2.9, $A \prec G$ since G belongs to $\mathcal{S}(\rho(A)) = \mathcal{S}(\{ALeft, ARight\}) = \{D, G, J, E, H, K\}$. Note that it is impossible to define the state precedence between J and K.*

Definition 2.7. *For all $S, S' \subseteq \mathcal{S}(H)$, the precedence between the sets S and S' is defined as:*

$$S \preceq S' \Leftrightarrow \forall s \in S, \exists s' \in S' : s \preceq s'$$

Knowing that the source restriction $sr \subseteq (\mathcal{S}(\rho(SRC(t))) \cup \{SRC(t)\})$ and that the target determinator $td \subseteq \mathcal{S}(\rho(TGT(t)))$, it is easy to understand the following definitions.

Definition 2.8 (Conflicting Transitions). *For $t, t' \in \mathcal{T}(H)$, t is conflicting with t' is written as $t\#t'$ and is defined as follows:*

$$t\#t' \Leftrightarrow \begin{array}{c} t \neq t' \\ \wedge \\ (SRC(t) \preceq SRC(t')) \vee (SRC(t') \preceq SRC(t)) \end{array}$$

Definition 2.9 (Priority Scheme). *Conflicts between $t, t' \in \mathcal{T}(H)$ are solved using the following priority scheme:*

$$t \sqsubset t' \quad \Leftrightarrow \quad \begin{array}{c} SRC(t) \prec SRC(t') \\ \vee \\ (SRC(t) = SRC(t')) \wedge (SR(t) \prec SR(t')) \end{array}$$

$$t \sqsubset_{=} t' \quad \Leftrightarrow \quad SRC(t) = SRC(t') \wedge SR(t) = SR(t')$$

$t \sqsubset t'$: t has lower priority than t'.
$t \sqsubset_{=} t'$: t and t' have equal priority.

Example 2.12. *Referring to Figures 2.8-2.9, we derive that $t2\#t5$ (Definition 2.8) and that $t2 \sqsubset t5$ (Definition 2.9).*

2.5.3 Operational Semantics

What we want to describe is the behavior of a statechart represented by its equivalent extended hierarchical automaton. An operational semantics focuses on the operations the system can perform — whether internally or interactively with some supersystem or the outside world.

The operational semantics of an extended hierarchical automaton will be defined as a Kripke structure, which is a set of states related by a total transition relation. Such a semantics computes necessarily infinite sequences of configurations; the computation doesn't stop if no further transition can be found. Instead, the next configuration will be exactly the same as the previous one i.e. self-transitions.

Definition 2.10 (Operational Semantics). *The operational semantics of an extended hierarchical automaton H is a Kripke structure $\mathcal{K} = (\boldsymbol{S}, \boldsymbol{s}_0, \overset{STEP}{\rightarrow})$ where*

- $\boldsymbol{S} = \mathcal{C}_H$ *is the set of statuses of* $\mathcal{K}$,
- $\boldsymbol{s}_0 = \mathcal{C}_{H_0} \in \boldsymbol{S}$ *is the set of initial statuses, and*
- $\overset{STEP}{\rightarrow}$ *is the transition relation defined in the sequel (Definition 2.16).*

Usually, the states are called *configurations* and the transition relation is called the *STEP relation*. The extended hierarchical automaton is supposed to react on messages from some supersystem or the outside world. In the definition of UML statechart diagrams the particular nature of the environment is not specified. Actually, we state it to be an *environment event generator* (EEG). The event generator creates the messages ($\in E_i$) from external objects the automaton responds to. Because there is no information included in the model, the event generator is non-deterministic.

It cannot be a suprise that a transition of $\mathcal{K}$ is a maximal set of non-conflicting transitions of the sequential automata of H which respect priorities. The maximal set of non-conflicting transitions is defined using the set of *enabled transitions*. A transition t is enabled in $\mathcal{C}$ if its guard expression is true, if it is triggered by the right event, and if all of its source states are active; it is executed if there is no higher priority transition that could preempt t.

Definition 2.11 (Enabled Transitions). *For a configuration $\mathcal{C}$, an event $e \in E$, the set of all enabled transitions, triggered by e, in $\mathcal{C}$ is defined as follows:*

$$ET(\mathcal{C})_e = \{t \in \mathcal{T}(H) \mid \{SRC(t)\} \cup SR(t) \subseteq \mathcal{C} \wedge EV(t) = e \wedge \mathcal{C} \models G(t)\}$$

Definition 2.12 (Maximal Enabled Transitions). *For a configuration $\mathcal{C}$, an event $e \in E$, the set of all maximal enabled transitions, triggered by e, in $\mathcal{C}$, written $MaxET(\mathcal{C})_e$, satisfies the following conditions:*

- $MaxET(\mathcal{C})_e \subseteq ET(\mathcal{C})_e$
- $\forall t, t' \in MaxET(\mathcal{C})_e : \neg(t \# t')$
- $\forall t \in MaxET(\mathcal{C})_e : \nexists t' \in ET(\mathcal{C})_e : t \sqsubset t'$

The role of the dispatcher is to find an event from E that can be accepted in the current configuration of the object. The dispatcher works on a First In First Out (FIFO) queue dedicated to flow and interaction events only. In the dispatching process, a higher priority is given to ε since triggering ε-transitions means that the current configuration is still not stable. Only when a stable configuration is reached, the dispatcher is allowed to dispatch an event from the queue.

Definition 2.13 (Dispatcher). *For a configuration $\mathcal{C}$, dispatch($\mathcal{C}$) returns the event ($\in E$) to be dispatched in $\mathcal{C}$:*

$$dispatch(\mathcal{C}) = \begin{cases} \varepsilon & \text{if } MaxET(\mathcal{C})_\varepsilon \neq \emptyset \\ q_e[0] & \text{if } MaxET(\mathcal{C})_{q_e[0]} \neq \emptyset \\ nil & \text{otherwise} \end{cases}$$

Once transitions are taken, a new configuration is entered and proper actions are performed. From now on, we refer to this as a *progress rule* and a *stuttering rule*. The first one is activated when there is a passage between two configurations due to the execution of some transition whereas the latter one has to be taken to guarantee that the transition relation is total i.e. each configuration must have, at any time, a subsequent configuration. The combination of both rules forms the transition relation of the Kripke structure $\mathcal{K}$ (Definition 2.16).

Definition 2.14 (Progress Rule). *Let $c_1 = \langle \zeta, v, q_e, hist \rangle$ and $c_2 = \langle \zeta', v', q_e', hist' \rangle$ be two configurations of $\mathcal{C}_H$, and $k = \mid MaxET(\mathcal{C})_{dispatch(\mathcal{C})} \mid$. If $dispatch(\mathcal{C}) \neq nil$ then the transition relation from c_1 to c_2 is defined as $c_1 \xrightarrow{prog}_{dispatch(\mathcal{C})} c_2$ with:*

$$\begin{array}{lcl} \zeta' & = & \zeta \setminus \bigcup_{1 \leq i \leq |k|} leaves(t_i) \cup \bigcup_{1 \leq i \leq |k|} enters(t_i) \\ v\text{'} & = & \textit{variable valuation after} \\ & & \textit{executing transitions of } MaxET(\mathcal{C})_{dispatch(\mathcal{C})} \\ q_e\text{'} & = & newQueue_e(\mathcal{C}) \\ hist\text{'} & = & newHist(\mathcal{C}) \end{array}$$

The function $newQueue_e$ returns a new queue of events. The head possibly is shifted out, and new events possibly are added to the end of the queue. Obviously, *newHist* maps automata to their last active substate. It is of course undefined, if no history state is attached to the automaton.

Definition 2.15 (Stuttering Rule). *Let $c_1 = \langle \zeta, v, q_e, hist \rangle$ and $c_2 = \langle \zeta', v', q'_e, hist' \rangle$ be two configurations of $\mathcal{C}_H$. If $dispatch(\mathcal{C}) = nil$ then the transition relation from c_1 to c_2 is defined as $c_1 \xrightarrow{stut} c_2$ with $c_1 = c_2$, i.e. configuration c_1 has to stutter.*

Definition 2.16 ($\overset{STEP}{\rightarrow}$). *The transition relation of a Kripke structure $\mathcal{K}$ is defined as follows:*

$$\overset{STEP}{\rightarrow} = \begin{cases} \xrightarrow{prog}_{\varepsilon} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=\varepsilon} \neq \emptyset \\ \xrightarrow{prog}_{q_e[0]} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=q_e[0]} \neq \emptyset \\ \xrightarrow{stut} & otherwise \end{cases}$$

The update of a configuration uses functions *enters* (Definition 2.17) and *leaves* (Definition 2.18). These functions compute the correct set of states to enter and to leave when a transition is executed.

Definition 2.17. *For a configuration $\mathcal{C}$, the set of states a transition t enters, when executed, is defined as follows:*

$$\{TGT(t)\} \cup TD(t) \cup (\bigcup_{s \in TD(t)} init(\rho(s)))$$

$$enters(t) = \cup \begin{cases} \bigcup\limits_{A \in \rho(TGT(t))} init(A) & \text{if } TD(t) = \emptyset \\ \bigcup\limits_{A \in \rho(TGT(t)), (x \in TD(t), s) \in \mathcal{S}(A), x \in \mathcal{S}(\rho(s))} s & \\ \cup \bigcup\limits_{A \in \rho(TGT(t)), (x \in TD(t)) \notin \mathcal{S}(A)} init(A) & otherwise \end{cases}$$

with

$$init(A) = \begin{cases} s^0_A \cup (\bigcup\limits_{A' \in \rho(s^0_A)} init(A')) & \text{if } hist(A) = nil \\ hist(A) \cup (\bigcup\limits_{A' \in \rho(hist(A))} init(A')) & otherwise \end{cases}$$

Obviously, when a transition t executes, its target states, $TGT(t)$ and $TD(t)$, are entered. Additionally, if $TD(t)$ is a composite state, either the system has to enter recursively predefined states — $\bigcup_{s \in TD(t)} init(\rho(s))$ — which are either default states or states that are marked by the history mapping. Besides these states, we also have to add some other states. If the target determinator of t is empty, then we have to enter $TGT(t)$ properly — $\bigcup_{A \in \rho(TGT(t))} init(A)$. Otherwise we add all those composite states ($\in TGT(t)$) that are either superstates of elements of $TD(t)$ or that are concurrent states of elements of $TD(t)$. An illustration of Definition 2.17 is given in the following example.

Example 2.13. *Let's return again to Figure 2.9 and considerer transition t_6. This transition enters a concurrent state in a very special way. From Definition 2.17 it follows that $TGT(t_6)$ and $TD(t_6)$ has to be included in $enters(t_6)$. Additionally, all the hierarchical states of region B that contain a member $TD(t_6)$ have to be included as well. Finally, we also have to add the initial states of region C (recursively) since t_6 enters a concurrent state and therefore enters each region. Summarized,*

$$\begin{aligned} enters(t_6) &= \{TGT(t_6)\} \cup TD(t_6) \cup \\ &\quad \cup \left\{ \begin{array}{l} \bigcup_{A \in \rho(TGT(t_6)), (x \in TD(t_6), s) \in \mathcal{S}(A), x \in \mathcal{S}(\rho(s))} s \\ \cup \bigcup_{A \in \rho(TGT(t_6)), (x \in TD(t_6)) \notin \mathcal{S}(A)} init(A) \end{array} \right. \\ &= \{A\} \cup \{G\} \cup \{D\} \cup \{E, H\} \end{aligned}$$

The set of states a transition leaves is much easier to compute. If a composite state is left, the state itself is left together with all of its active descendants.

Definition 2.18. *For a configuration $\mathcal{C}$, the set of states a transition t leaves, when executed, is defined as follows:*

$$leaves(t) = \{SRC(t)\} \cup \{s \in \mathcal{S}(\rho(SRC(t))) \mid s \in \mathcal{C}\}$$

2.5.4 Summary

The operational semantics of UML state machines is defined as a Kripke structure. Such a structure forms the basic for embedding statecharts into temporal logic, in particular, the symbolic execution technique integrated in the model checker CaSMV. In fact, the operational semantics is defined as a collection of run-to-completion steps between statechart configurations. Each

RTC-step is formally based upon two update rules: the progress rule and the stuttering rule. Special about the progress rule is that it first computes, based on priority rules, a maximally consistent set of enabled transitions for a given event and then executes this set of transitions using functions like *enters* and *leaves*.

2.6 Related Work

We will discuss related work dealing with formal semantics definitions of UML statecharts.

The work of Latella et al. [73] has been one starting point of our work. In contrast to their work, our approach includes the history mechanism, supports actions, and integrates stuttering rules in the formal semantics definitions. We let us inspire by [36], to reduce the complexity of the semantics of [73] to a semantics that has a quite clear connection by the formalism accepted by CaSMV.

Paltor and Lilius [99] as well as Kwon [67] define an operational semantics for UML statecharts in terms of rewrite rules. Since they do not use a structured approach like ours, their semantics does not offer the same level of clarity.

Both [5] and [113] have proposed a formal semantics definition for UML statecharts in the PVS specification language. But to the best of my knowledge, they do not handle events adequately enough i.e. the order of event arrivals is neglected.

For a complete list of formal semantics definitions for UML statecharts, we refer to [31, 114].

CHAPTER 3

The Model of a Standalone Statechart

It claims to be fully automatic, but actually,
you have to push this little button here.
Gentleman John Killian.

The ever increasing complexity of embedded systems poses a challenge in verifying their correctness. For such safety-critical applications an approach is definitely needed to validate the complexity of the behavioral designs at a higher level of abstraction. With formal verification we verify that every possible behavior of the target system satisfies the specification.

By applying verification techniques we can reduce the cost of development much earlier in the lifecycle while defects are relatively inexpensive to correct. Applying formal techniques in the development of embedded systems relies on a model of the system, a description of the requirements of the system and a technique to check that the requirements are satisfied by the model.

In this chapter, we propose a UML-based verification method that checks whether the conditions on the evolution of the embedded system are met by the model. The general approach is to abstract some kind of model from the code (i.e. statechart diagrams), against which the software can be proven through the use of appropriate mathematics. Model checking is an example of such a mathematics, and using the Cadence version of SMV (CaSMV) [86] guarantees that the verification process can be performed fully automatically.

This chapter considers embedded systems whose behaviors are expressed with standalone statechart diagrams. The transformation from the statechart to a model, required by the model checking technique and CaSMV, is heavily addressed here. Additionally, we introduce some basic concepts of model checking and outline the place of the statechart transformation in our methodology.

3.1 Model Checking with Cadence SMV

In a nutshell, model checking is a technique for automated correctness verification of safety-critical reactive systems. A reactive system is a system consisting of several components designed to continuously interact with one another and with the system's environment. On the other hand they are also control-oriented. Therefore, model checking can be applied to analyse embedded systems, whose behavior is expressed within a standalone UML statechart diagram.

From a logical viewpoint, the general approach of the model checking technique (see Figure 1.4 and Section 1.6.3) consists of three steps. In the first step, the system is expressed as a semantic *model* $\mathcal{M}$. The model $\mathcal{M}$ is a finite model that captures the intuition about the behavior of a reactive system; given as a *Kripke structure* [26]. The second step defines the specification of a property φ expected of the system. The property φ is a set of desired behaviors in time, and is therefore a logical formula defined in a temporal logic. In the last step, model checking amounts to determining the truth of formulas in models, i.e. whether $\mathcal{M} \models \varphi$.

3.1.1 Kripke Model

A Kripke model $\mathcal{M}$ (Definition 3.1) is a modeling formalism independent way of representing the behavior of a system. A Kripke model is basically a graph having the reachable states of the system as nodes and the state transitions of the system as edges. It is a non-deterministic finite state machine whose states are labeled with boolean variables, which are the evaluations of expressions in that state [17].

To obtain a model $\mathcal{M}$ we need a set of atomic propositions AP, which denote the properties of individual states we are interested in. Additionally, a set of transitions between states, and a function that labels each state with a state of properties that holds in that state.

Definition 3.1 (Kripke Model). *A Kripke Model $\mathcal{M}$ over a set of atomic propositions AP is a tuple $\mathcal{M} = (\sigma, \sigma_0, \delta, \lambda)$. It consists of a Kripke Structure $(\sigma, \sigma_0, \delta)$ and a labeling function λ where*

- *σ is the set of states.*
- *$\sigma_0 \subseteq \sigma$ is the set of initial states.*
- *$\delta \subseteq \sigma \times \sigma$ is the transition relation that must be total, that is, for every state $s \in \sigma$ there is a state $s' \in \sigma$ such that $\delta(s, s')$.*

- $\lambda : \sigma \rightarrow 2^{AP}$ *is a function that labels each state with the state of atomic propositions that are true in that state (e.g.* $a = 1$*).*

Note that in Kripke models deadlock states are disallowed. This is for technical reasons (it simplifies the theory somewhat). Instead, in the corresponding Kripke model we add an edge from the deadlock state back to itself.

As can be directly seen from the definition, Kripke models have a close relationship with automata. But there are some changes. To give an example, labeling is on states instead of having labels on arcs. To give another example, there is no definition of final states.

Example 3.1. *Figure 3.1 is the representation of a fairly small Kripke model. We have:* $\sigma = \{S0, S1\}$*;* $\sigma_0 = \{S0\}$*;* $\delta = \{(S0, S1), (S1, S0)\}$*;* $\lambda(S0) = \{x = 0, y = 1\}$*;* $\lambda(S1) = \{x = 1, y = 0\}$.

Figure 3.1: A Small Kripke Model

Cadence SMV [86] is a formal verification system for designs, based on a technique called symbolic model checking [87]. It verifies that every possible behavior of the target system satisfies the specification.

The required model for CaSMV is, of course, a Kripke model, whose state is defined by a collection of state variables. The transition behavior and its initial state(s) are determined by a collection of parallel assignments which, when solved, tell us what state transitions the program can make.

The required specification for CaSMV is a collection of properties in temporal logic. A property can be as simple as a statement that a particular pair of signals are never asserted at the same time, or it might state some complex relationship in the values or timing of the signals.

CaSMV is quite effective in automatically verifying properties of combinational logic and interacting finite state machines. Sometimes, when the checking of a property fails, the tool will automatically produce a counter-example. This is a behavioral trace of the finite state machines that violates the specified property. Thus making CaSMV a very effective debugging tool, as well as a formal verification system.

3.1.2 Introductory Example

Rather than giving a complete overview of the syntax and the semantics of the CaSMV language, let us first consider a simple example that illustrates the basic concepts. The following presentation is adapted from the SMV manual [87]. Further details on the CaSMV system are in [89, 90].

Consider the CaSMV program in Code Listing 3.1. The first part defines the Kripke model. The space of states σ of the Kripke model is determined by the declarations of the state variables (in the example `request` and `state`). The variable `request` is declared to be of (predefined) type boolean. This means that it can assume the (integer) values 0 and 1. The variable `state` is a scalar variable, which can take the symbolic values `ready` or `busy`. Thus, a state of σ is defined by mapping the state variables to a value from the given domains.

Code Listing 3.1 An Example CaSMV Program

```
module main() {

  request: boolean;
  state  : {ready, busy};

  init(state) := ready;
  next(state) := case {
                   state = ready & request: busy;
                   default                : {ready, busy};
                 };

  property: SPEC AG(request -> AF state = busy);

}
```

The following assignment sets the initial value of the variable `state` to `ready`. The initial value of request is completely unspecified, i.e. it can be either 0 or 1. Therefore, the initial set of states σ_0 is given by these `init`-statements. Be aware of the fact that initial-state assignements are simultaneously executed at the start.

The transition relation δ of the Kripke model is expressed by defining the value of variables in the next state (i.e. after each transition), given the value of variables in the current states (i.e. before the transition). The case segment sets the next value of the variable `state` to the value `busy` (after the column) if its current value is `ready` and `request` is 1 (i.e. true). Otherwise

the next value for `state` can be any in the set {`ready,busy`}. Obviously, next-state assignments are simultaneously executed once per cyle.

The variable `request` is not assigned. This means that there are no constraints on its values, and thus it can assume any value, which is thus an unconstrained input to the system.

Specifications can be expressed in CTL as well as in LTL. The keyword `SPEC` is followed by a CTL formula, that is intended to be checked for truth in the Kripke model defined above. The intuitive reading of the formula is that every time `request` is true, then in all possible future evolutions, eventually `state` must become `busy`.

The resulting transition system is visualized in Figure 3.2. Obviously, the elements of the Kripke model are as follows: $\sigma = \{S0, S1, S2, S3\}$; $\sigma_0 = \{S0, S1\}$; $\delta = \{(S0, S1), (S0, S2), (S0, S3), (S0, S0), \ldots\}$; $\lambda(S0) = \{state = ready, request = 0\}$; ....

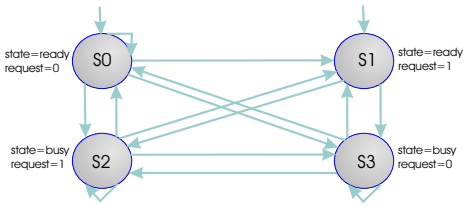


Figure 3.2: Kripke Model of the Introductory Example

3.2 Methodology

We suppose that the behavior of embedded systems is specified within a single statechart diagram. A prerequisite to formally verify behavioral properties is to map the statechart diagram to a formal semantic model. Naturally, the semantic model should satisfy the UML semantics of statecharts. Model checking the behavior of only a single object (i.e. a single statechart), requires an interpretation of the statecharts model as a Kripke model. It is the responsibility of our methodology to construct a semantic model, required by the model checker CaSMV, for such embedded systems. In fact, at this point, we start building the architecture of our *simple-to-use* tool (Figure 3.3) that helps in the formal verification of embedded systems design. Chapter 9 applies the verification methodology to a far more complex example, which is considered as an industrial benchmark.

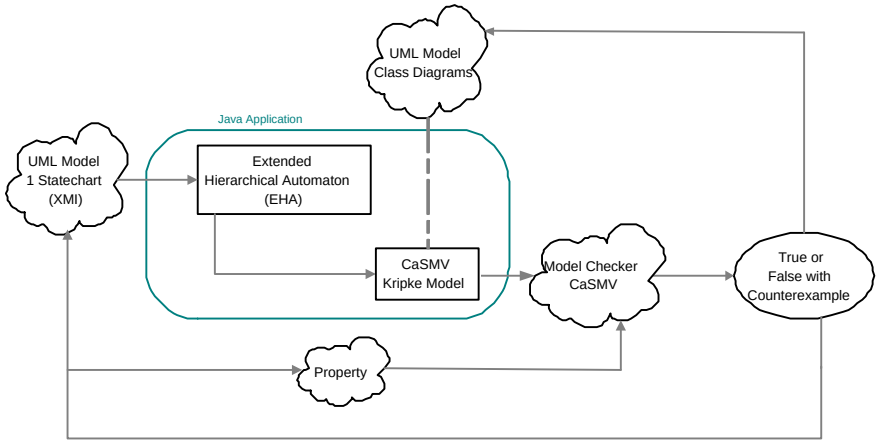


Figure 3.3: Methodology & Tool Architecture

The tool input is a UML specification which has been formatted using the XMI exchange syntax. For verification purposes, the XML representation of the statechart diagram is post-processed by a Java application [53, 122] that generates an EHA; as a semantically equivalent formal model. In a next step, and based on the formal operational semantics, the same Java application automatically transforms the EHA to the requested CaSMV model[1].

To summarize things, our tool provides the mapping from a statechart diagram to a CaSMV model (= Kripke model), through an intermediate model, namely an extended hierarchical automaton. The CaSMV model satisfies the semantics defined in Chapter 2, Section 2.5. Next to a statechart diagram, class diagrams are taken into account as well when transforming the embedded behavior from UML to CaSMV. These diagrams provide important additional information concerning some elements that make up the semantic model e.g. types of variables or initial values of variables.

3.3 Motivating Example

We will use the *coffee vending machine* [56] as an example in order to show how CaSMV models are constructed. Both because it is a well-known example of an embedded system, and because it incorporates in its specification both reactiveness and control-oriented features.

Assume that we have a statechart modeling a coffee vending machine

[1]See also Section 3.8.

(Figure 3.4). Then a possible event is a user inserting a coin which, when it occurs, leads to a state change of the automaton. Obviously, the user forms a part of the environment of the machine. After inserting some coins and pushing a button, additional state changes and actions occur and finally coffee is offered to the user.

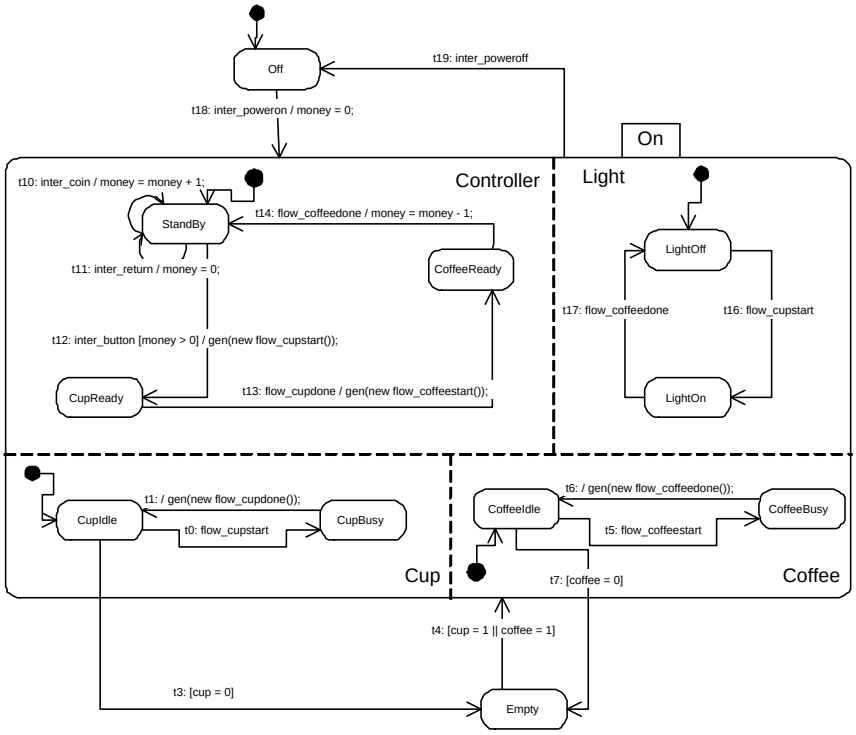


Figure 3.4: Statechart Diagram for a Coffee Vending Machine

The extended hierarchical automaton, primarily used in the following sections, is given in Figure 3.5.

3.4 From a Statechart to a Kripke Model

As we have mentioned, model checking a statechart diagram requires a mapping from the statechart to a Kripke model. The automatic transformation of a statechart to a Kripke model $\mathcal{M}$ (Definition 3.1), through an EHA, has to respect the automaton's operational semantics, which is defined as a

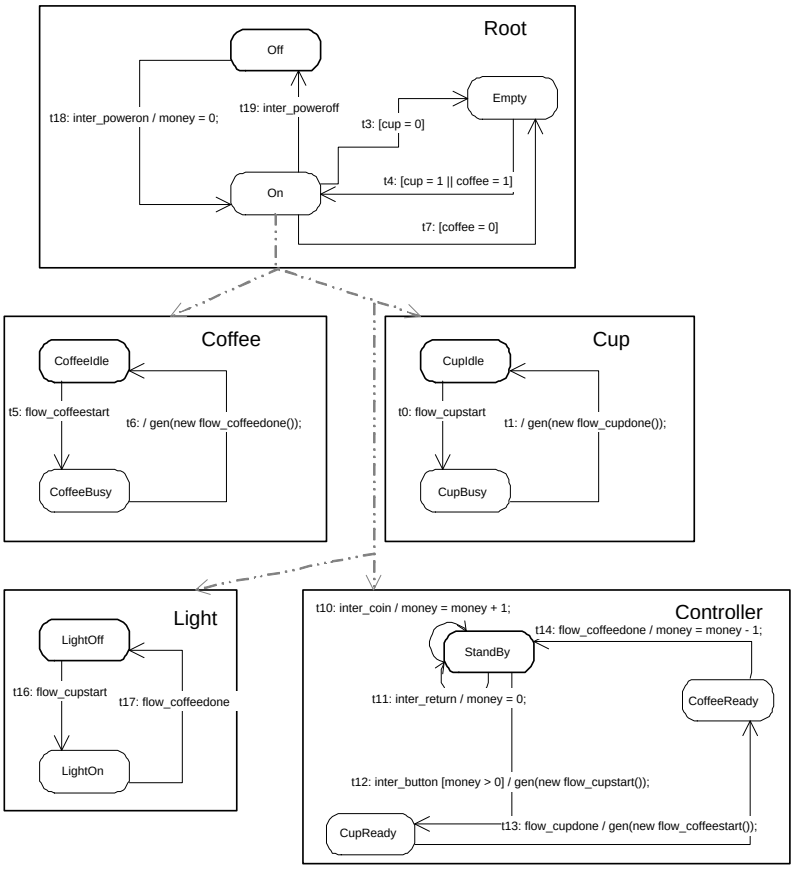


Figure 3.5: Extended Hierarchical Automaton for a Coffee Vending Machine

Kripke structure $\mathcal{K}$ (Definition 2.10). Not surprisingly, $\mathcal{K}$ has to be correctly integrated into $\mathcal{M}$. The integration is defined as follows:

$$\begin{array}{lcccc} \mathcal{K} = & (\mathcal{C}_H, & \mathcal{C}_{H_0}, & \overset{STEP}{\rightarrow}) \\ & \downarrow & \downarrow & \downarrow \\ \mathcal{M} = & (\sigma, & \sigma_0, & \delta, \quad \lambda) \end{array}$$

Let's explain the integration into more detail. Intuitively, you must already know that a state of σ is a snapshot of all the variables of the given system. Based on this, we will map configurations of the complete EHA to states of σ ($\mathcal{C}_H \longmapsto \sigma$). Since each configuration (Definition 2.5) is a quadruple *(states,variables, event_queue,history mapping)*, a state of σ is a quadruple

as well. In Section 3.5.4, it is explained that the *history mapping* is treated in a very special way. As a consequence, a state of σ will be defined as a triple instead of as a quadruple: $(\text{states, variables, event\_queue})_H \longmapsto \sigma$.

The set of initial states σ_0 is given by the set of initial configurations which is derived from the initial configuration of the set of sequential automata in a top-down manner starting from the root automaton ($\mathcal{C}_{H_0} \longmapsto \sigma_0$).

The transition relation δ will correspond to the STEP relation $\overset{STEP}{\rightarrow}$. Therefore, the complete execution semantics of statecharts will be covered by δ ($\overset{STEP}{\rightarrow} \longmapsto \delta$).

The labeling function λ immediately follows from the above three defined mappings. In fact, you may notice that a state is not explicitly labeled, but it is implicitly labeled by its corresponding state-variable valuations.

The construction of the model $\mathcal{M}$ is split up into three different blocks i.e. Section 3.5 defines the block of state variables (since these are used to define σ), Section 3.6 specifies the block of initial states, and finally Section 3.7 carries out the block that covers the transition relation.

3.5 The Set of States σ

A run-to-completion step is defined as the passage between two stable state configurations, possibly going through several instable ones. Not surprisingly, σ will be defined as the set of statuses $\mathbf{S} = \mathcal{C}_H$. Therefore, each element that belongs to a configuration $\mathcal{C}$ (Definition 2.5) will correspond to a suitable CaSMV variable. Declarations of these variables determine the space of states σ.

3.5.1 Local States

A configuration $\mathcal{C}$ denotes a global state of an EHA, and is composed of local states of component sequential automata (Definition 2.10). To correctly control the evolution of a state machine, the state it is in at any given moment must be known. This is achieved by using a separate variable to store this information for each machine.

In addition, the fact that combined states, both sequential and concurrent, may appear within a machine means that additional variables are needed in order to deal with submachines. This will be dealt by following the same reasoning as for the main machine. The latter will, of course, correspond with the root state.

Specifically, each sequential automaton of the EHA tree is represented by a CaSMV scalar variable (finite domain). Rule 3.1 defines the possible values of such variables. Note that the `NotActive` value is optional since it can be possible that the machine continuously is in an automaton during its evolution (e.g. no final state is reached). The value `StateMachineError` is only added to the scalar variable of the top automaton to indicate whether or not a fault has occurred during the verification process.

Rule 3.1 (Automaton Declaration). *The root automaton $A_0 \in F$ is represented by a scalar variable:*

$$st\_Root : \{\sigma_{A_0}, [NotActive], StateMachineError\};$$

Each automaton $A \in F \setminus \{A_0\}$ is represented by an enumerated variable:

$$st\_A : \{\sigma_A, [NotActive]\};$$

An illustration of such declarations is given in Code Listing 3.2. At this point the usefullness of extended hierarchical automata must become clear.

Code Listing 3.2 Local State Declarations

```
st_Root      : {Off, Empty, On, StateMachineError};
st_Coffee    : {CoffeeIdle, CoffeeBusy, NotActive};
st_Light     : {LightOn, LightOff, NotActive};
st_Cup       : {CupBusy, CupIdle, NotActive};
st_Controller: {StandBy, CupReady, CoffeeReady, NotActive};
```

On the other hand, flattening the hierarchy is a very straightforward way to represent a statechart as a transition system. But our representation avoids such an approach, since this can lead to an exponential blow-up in size, particularly when there is a lot of sharing of states and transitions.

3.5.2 Variable Valuation

Each configuration $\mathcal{C}$, no matter if it is stable or not, has specific values for the variables (attributes) used by the embedded system (Definition 2.10). And the evolution of the machine can lead to different actions such as modifying the value of several attributes. Therefore, attributes of the system has to be modeled in $\mathcal{M}$ as well.

All attributes, whose value changes due to the evolution of the statechart, are gathered into *defined types* (see Rule 3.2). A *type definition* is a special

kind of module declaration[2] with no parameters, and a slightly different syntax.

Rule 3.2 (Attribute Declaration). *Let Attr be the set of attributes whose values change due to the evolution of the machine. Each* $attr_i \in Attr$ *is declared inside a type definition as follows:*

$$typedef\ attributes\ struct\ \{attr_i\ :\ finite\ type\ of\ attr_i;\ \}$$

A state variable is created by instantiating the type definition:

$$attr : attributes;$$

Knowledge about attributes and their corresponding types is retrieved from the class diagram. Typed reasoning and attribute domain information can be used to inject type specific operations and constraints in the CaSMV model checker. Attributes of the coffee vending machine are modeled as shown in Code Listing 3.3.

Code Listing 3.3 Attribute Declarations

```
typdef attributes struct {
  money : 0..20;
  cup   : boolean;
  coffee: boolean;
}
```

3.5.3 Event Queue

The actual state of a state machine is also given by the contents of its event queue (Definition 2.10). The event queue holds the events that have not yet been handled by the machine. The finite set of events E of an EHA (Definition 2.3) holds three types of events: flow events, interaction events and the null event. However, in the model $\mathcal{M}$ the event queue will store interaction and flow events only. As we will see later, the null event will be treated in a special way.

This time, an `array` is used to model the event queue. Each array is bounded since $\mathcal{M}$ has to be a finite model, and therefore only finite types

[2] A *module* is a bundle of definitions (type declarations and assignments) that can be reused. When creating an *instance* of the module, actual expressions are plugged in for the formal parameters, thus linking the module instance into the program. E.g. `module counter_bit(carry_in, clear, bit_out, carry_out){...}`

can be added to the model. To be able to store interaction and flow events, the type of the array is an enumerated type. Naturally, a `NotDefined` value is used to specify empty positions. Additionally, a pointer to the first free queue position, and a overflow indicator is inserted into the model. The modeling of the event queue is more formally defined in Rule 3.3, and an illustration is given in Code Listing 3.4.

Rule 3.3 (Event Queue Declaration). *The event queue of a configuration $\mathcal{C}$ is modeled as follows:*

$$event\_queue:\ array\ 0..(SIZE\ -\ 1)\ of\ \{E_i, E_f, NotDefined\};$$

The pointer to the first empty queue position is modeled as follows:

$$event\_tail:\ 0..(SIZE\text{-}1);$$

The queue overflow indicator is declared as follows:

$$event\_overflow:\ boolean;$$

Code Listing 3.4 Event Queue Declarations

```
event_queue    : array 0..(SIZE - 1) of
                    {  inter_poweron, inter_poweroff, inter_coin,
                       inter_return, inter_button,
                       flow_coffeestart, flow_cupstart,
                       flow_coffeedone, flow_cupdone, NotDefined
                    };
event_tail     : 0..(SIZE-1);
event_overflow: boolean;
```

3.5.4 History Mapping

The history mapping of a configuration, that attaches the last visited substate to each sequential automaton, is not modeled explicitly as a state variable. History states are treated in a special way at the moment the transition behavior is constructed. Thus, the mapping will be implicitly integrated in the model. How this is actually achieved, will be shown further on (Section 3.7).

3.6 Initial States σ_0

The model checker CaSMV starts the reachability analysis by initializing the set of state variables. Initializing variables in CaSMV is done using the `init` operator.

3.6.1 Local States

Each sequential composite state is initialized to its default state because the default state is the first state entered when its containing state becomes active. However, only the sequential composite states that the machine resides in before execution starts, are initialized. The initial value of the other states is set to `NotActive`; they are not supposed to be active at the very beginning of a run.

Rule 3.4 (Automaton Initialization). *The root automaton $A_0 \in F$ is initialized in $\mathcal{M}$ using a trivial program statement:*

$$init(st\_Root) = s^0_{Root};$$

Each automaton $A \in F \setminus \{A_0\}$ is initialized as follows:

$$init(st\_A) = \begin{cases} s^0_A; & if\ A \in default(s^0_{Root}) \\ NotActive; & otherwise \end{cases}$$

with

$$default(s) = \rho(s) \cup \bigcup_{A \in \rho(s)} default(s^0_A)$$

Rule 3.4 is quite useful to initialize our sample embedded system, as shown in Code Listing 3.5. When the machine comes into existence, the default state `Off` is the only state that is entered immediately.

Code Listing 3.5 Automaton Initialization

```
init(st_Root)      := Off;
init(st_Coffee)    := NotActive;
init(st_Light)     := NotActive;
init(st_Cup)       := NotActive;
init(st_Controller):= NotActive;
```

3.6.2 Variable Valuation

Each attribute is initialized to its default value set in the UML class diagram or to the value assigned in a default state that is active when the machine becomes alive. If the initial value of an attribute is not defined, we omit an initial assignment. As a consequence CaSMV will explore all possible initial values.

3.6.3 Event Queue

There are two kinds of models: the closed model and the open model. In a *closed model*, the machine does not interact with the outside world i.e. $E_i = \emptyset$. In contrast, some models need external stimuli such as the coffee vending machine. *Open models*, that react to messages sent by external entities, cannot be verified with a model checker since these messages are not generated by any element of the model. They can only be analysed when the events are created by an *event generator*. Because there is no information included in the model, the event generator is non-deterministic. When located in some status, the external events that may enable some transitions in the current status are selected for once. The reason is that for a single statechart, the behavior of the environment is completely unpredictable. Then all the possible subsequent statuses are found, and which status will be reached is non-deterministic.

Rule 3.5 (Event Queue Initialization). *Let $E_{\mathcal{C}_0}$ be the set of external stimuli the initial configuration $\mathcal{C}_0$ responds to. Then, the event queue is initialized as follows:*

$$
\textit{if } E_{\mathcal{C}_0} = \emptyset \textit{ then } \begin{cases} \textit{init(event\_queue)} & := & [NotDefined : i = 0..(SIZE-1)]; \\ \textit{init(event\_tail)} & := & 0; \\ \textit{init(event\_overflow)} & := & 0; \end{cases}
$$

$$
\textit{otherwise} \quad \begin{cases} \textit{init(event\_queue[0])} & := & \{E_{\mathcal{C}_0}\}; \\ \textit{init(event\_queue[1])} & := & \textit{NotDefined}; \\ \ldots & := & \ldots; \\ \textit{init(event\_queue[SIZE-1]))} & := & \textit{NotDefined}; \\ \textit{init(event\_tail)} & := & 1; \\ \textit{init(event\_overflow)} & := & 0; \end{cases}
$$

The initialization of the event queue is dependent on the presence of interaction events in the initial configuration of the extended hierarchical automaton. The queue is empty if the initial configuration does not respond to the messages sent by the outside world. Otherwise the event generator adds one of the messages non-deterministically to the first position of the queue. Rule 3.5 defines the way the event queue of the coffee vending machine needs to be initialized (Code Listing 3.6).

Code Listing 3.6 Event Queue Initialization

```
init(event_queue[0])      := {inter_poweron};
init(event_queue[1])      := NotDefined;
...;
init(event_queue[SIZE-1]) := NotDefined;

init(event_tail)     := 1;
init(event_overflow) := 0;
```

Why does the presence of external stimuli imply different CaSMV initializations? The reason is that we are restricted to the *single assignment* rule, provided by CaSMV. Each variable can be assigned only once in a program. The rule is that we may either assign a value to `x`, or to `init(x)` and `next(x)`, but not both. Tables 3.1 and 3.2 illustrate legal assignments and illegal ones respectively [89, 90].

x:= 0;	next(x):= 1;
init(x):= 0;	init(x):= 0; next(x):= 1;

Table 3.1: Legal CaSMV assignments

Assigning an array reference with a variable index counts as assigning every element in the array, as far as the single assignment rule is concerned [89, 90]. Thus for example,

```
x[0]         := 0;
x[count + 1]:= 1;
```

is a violation of the single assignment rule, if `count` is not a constant. This is because CaSMV cannot determine at compile time that `count + 1` is not equal to `0`.

x:= 0; x:= 1;	next(x):= 1; next(x):= 0;
x:= 1; init(x):= 0;	x := 1; next(x):= 0;

Table 3.2: Illegal CaSMV assignments

3.7 Transition Behavior δ

The transition behavior of $\mathcal{M}$ is formulated using the STEP relation of $\mathcal{K}$ (Definition 2.16). This way, $\mathcal{M}$ satisfies the operational semantics completely. Since the STEP relation goes from one configuration to another, it is necessary to define the transition behavior for each element of such configurations.

Section 3.7.1 generally outlines how $\overset{STEP}{\rightarrow}$ is modeled in CaSMV. Sections 3.7.2-3.7.3 define two important encoding mechanisms. Sections 3.7.4-3.7.6 define the constructs that are needed to guarantee some evolution while the statechart is verified against expectation properties. Finally, Section 3.7.7 points out the way the statechart stutters during the verification process.

3.7.1 STEP Relation

To recap, the general structure of a step can be seen as a combination of two closely related phases and a stutter rule (Definition 2.16). The first phase checks whether an instable state configuration can evolve on its own. This means that null-triggered transitions that may fire are identified and executed. Similarly, the second phase checks whether a stable configuration can evolve by dispatching an event from the environment. Like in the first phase, transitions that may fire are identified and executed.

Rule 3.6 (Step Relation Representation). *Let $T_{if} \subseteq \mathcal{T}(A_0)$ be the set of all transitions triggered by an external event ($\in E_i \cup E_f$). Let $T_\varepsilon \subseteq \mathcal{T}(A_0)$ be the set of all null-triggered transitions. Then, the STEP relation of $\mathcal{K}$ is*

inserted into $\mathcal{M}$ *as follows:*

$$
\begin{aligned}
&\mathit{progress\_auto} &&:= \bigvee_{t \in T_\varepsilon} t;\\
&\mathit{progress\_trigger} &&:= \bigvee_{t \in T_{if}} t;
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{case}\ \{\\
&\quad \mathit{progress\_auto}\ \&\ {\sim}\mathit{error} : \{\ldots\};\\
&\quad \mathit{progress\_trigger}\ \&\ {\sim}\mathit{error} : \{\ldots\};\\
&\quad \mathit{error} : \{\ldots\};\\
&\quad \mathit{default} : \{\ldots\};\\
&\};
\end{aligned}
$$

Rule 3.6 exploits the power of the branch statement to define the correct execution order between the several phases of the $STEP$ relation in the transition relation. For example, the second branch statement is only taken when the state machine is in a stable configuration (represented by `progress_trigger`). As long as the statechart is in some intermediate and inconsistent situation (represented by `progress_auto`), the first branch statement will be taken.

To acquire a total transition relation, an additional stutter rule is added to the $STEP$ relation of $\mathcal{K}$ when inserted into the model $\mathcal{M}$. Stuttering rules provide that finite runs are always interpreted as special cases of infinite runs. A configuration may stutter due to two reasons:

- At the moment an error has occurred, the state configuration is forced to stutter in a so-called *exception configuration*.
- Due to the *lack of progress*, the state configuration remains the same. This happens when no transitions can be identified to fire. This rule prevents the model to behave incorrectly at the moment a (in)stable configuration cannot evolve anymore.

It is clear that the statechart transitions are divided into two sets. One set contains all the transitions without a trigger while the other set consists of the transitions with a trigger. The progress rule (Definition 2.14) indicates that we have to know, at each moment, whether a state configuration can still evolve or not. This is the responsibility of `progress` variables. The *valuation* of both `progress` variables depends on the enabledness conditions of transitions, and a *truth value* indicates progress in the current state configuration. Such variables are *macro variables*[3]; they will not be added to

[3]Macro variables make descriptions more concise.

the state space of $\mathcal{M}$. Whenever a progress variable occurs in an expression, it is replaced by the *valuation* of the corresponding disjunction (not the disjunction itself!).

3.7.2 Encoding Active States

Since the run-to-completion step works with configurations — sets of active states — it is necessary to specify for each state CaSMV conditions that make the state active. Such conditions (Rule 3.7) are again macro variables. Clearly, the conditions exploit the structure of an extended hierarchical automaton; a state can only be active if its parent state is active as well.

Rule 3.7 (Active State Conditions). *For the root automaton $A_0 \in F$, the conditions that make the states $s \in \sigma_{A_0}$ active are defined as follows:*

$$\begin{aligned} &in\_s : boolean; \\ &in\_s := (st\_Root = s); \end{aligned}$$

For each automaton $A \in F \setminus \{A_0\}$, and a state $s \in \mathcal{S}(A_0), A \in \rho(s)$ the conditions that make the states $s' \in \sigma_A$ active are defined as follows:

$$\begin{aligned} &in\_s' : boolean; \\ &in\_s' := in\_s \;\&(\; st\_A = s'); \end{aligned}$$

Rule 3.7 applied on the coffee vending machine gives the result as shown in Code Listing 3.7. The `Empty` state is active iff the machine resides in the state; thus the value of `st_Root` is `Empty`. Analogously, the light is off iff the machine is in the concurrent state `On` and iff the region `Light` resides in the `LightOff` state.

3.7.3 Encoding Enabled Transitions

Active state conditions simplify the representation of enabled transitions, as defined by Rule 3.8 and illustrated in Code Listing 3.8. A transition t is enabled (Definition 2.11) in $\mathcal{C}$ if all of its source states are active, if its guard expression is true and if its trigger is offered by the environment and is currently being dispatched (the head of the FIFO queue). Naturally, the guard and the trigger are optional elements in these conditions.

Code Listing 3.7 Active State Conditions

```
in_Off, in_Empty, in_On: boolean;
in_Off  := (st_Root = Off);
in_Empty:= (st_Root = Empty);
in_On   := (st_Root = On);

in_CoffeeIdle, in_CoffeeBusy: boolean;
in_CoffeeIdle:= in_On & (st_Coffee = CoffeeIdle);
in_CoffeeBusy:= in_On & (st_Coffee = CoffeeBusy);

in_LightOn, in_LightOff: boolean;
in_LightOn := in_On & (st_Light = LightOn);
in_LightOff:= in_On & (st_Light = LightOff);
...;
```

Rule 3.8 (Enabled Transition Representation). *The enabledness condition of each transition $t \subseteq \mathcal{T}(A_0)$ is specified as follows:*

t : *boolean*;
$t :=$ *in_SR(t)* [& *G(t)*][& *event_queue[0]* = *EV(t)*];

with

$$in\_SR(t) := \bigwedge_{s \in SR(t)} in\_s;$$

As you can see, we haven't used $SRC(t)$ in the enabledness condition. This is because we have constructed the source restriction in such a way that $SR(t) \subseteq (\mathcal{S}(\rho(SRC(t))) \cup \{SRC(t)\})$. For example, consider transition `t3` with `SRC(t3) = On` and `SR(t3) = CupIdle`. The source restriction of `t3` will only be active if its parent state is active as well. As another example, let's look at transition `t5`. Now, `SRC(t5) = SR(t5) = CoffeeIdle` thus it is again useless to explicitly add `SRC(t5)` in the enabledness condition.

3.7.4 Local States, Priority Scheme, History Mapping

In a step, the system reacts to stimuli by propagating to other states i.e. one configuration changes to another configuration. Obviously, the updating to a new configuration depends on the transitions that are enabled for firing since the selected transitions cause the current active states to change to the transitions destination states.

Code Listing 3.8 Enabledness Conditions

```
t3, t4, t7, t18, t19 : boolean;
t3 := in_CupIdle & (cup = 0);
t4 := in_Empty & (cup = 1 || coffee = 1);
t7 := in_CoffeeIdle & (coffee = 0);
t18:= in_Off & event_queue[0] = inter_poweron;
t19:= in_On & event_queue[0] = inter_poweroff;

t5, t6 : boolean;
t5:= in_CoffeeIdle & event_queue[0] = flow_coffeestart;
t6:= in_CoffeeBusy;

t16, t17 : boolean;
t16:= in_LightOff & event_queue[0] = flow_cupstart;
t17:= in_LightOn & event_queue[0] = flow_coffeedone;
...;
```

As mentioned, each state variable represents a sequential automaton. In a sequential automaton there can be, at any time, at most one transition being enabled for firing. The collection of all these selected transitions make up the set of maximal enabled transitions (Definition 2.12).

To model state changes (for each state variable) in CaSMV, a collection of parallel `next` statements is used. A `next` statement defines what the value of a state variable is, one time unit further, based on Definitions 2.17- 2.18. Each `next` statement (Rule 3.9) consists of the identification of the transitions that may fire based on the priority scheme (Definition 2.9).

Rule 3.9 (State Change Representation). *The state changes for each*

automaton $A \in F$ are defined as follows:

$$
\begin{array}{l}
\mathit{next(st\_A)} := \\
\quad \mathit{case}\ \{ \\
\quad\quad \text{- - } \mathit{entering\ transitions} \\
\quad\quad t_i : \begin{cases} s_A^0; & \mathit{if\ no\ history\ state},\ TGT(t_i) = s,\ TD(t_i) = \emptyset \\ k; & \mathit{if}\ TGT(t_i) = s,\ k \in TD(t_i),\ k \in \sigma_A \\ k'; & \mathit{if}\ TGT(t_i) = s,\ k' \in \sigma_A,\ \exists k \in TD(t_i) \wedge k \in \mathcal{S}(\rho(k')) \end{cases} \\
\\
\quad\quad \text{- - } \mathit{deepest\ transitions\ followed\ by\ higher\text{-}level\ transitions} \\
\quad\quad t_j : \begin{cases} TGT(t_j); & \mathit{if}\ TGT(t_j) \in \sigma_A \\ st_A; & \mathit{if}\ TGT(t_j) \in \mathcal{S}(A) \setminus \sigma_A \\ NotActive; & \mathit{if\ no\ history\ state},\ TGT(t_j) \notin \mathcal{S}(A) \\ st_A; & \mathit{if\ history\ state},\ TGT(t_j) \notin \mathcal{S}(A) \end{cases} \\
\\
\quad\quad \text{- - } \mathit{equal\ conflicting\ transitions} \\
\quad\quad \mathit{choose\_in\_} \ldots \mathit{\_\ event} = T_k : \\
\qquad\qquad\qquad \begin{cases} TGT(t_k); & \mathit{if}\ TGT(t_k) \in \sigma_A \\ st_A; & \mathit{if}\ TGT(t_k) \in \mathcal{S}(A) \setminus \sigma_A \\ NotActive; & \mathit{if\ no\ history\ state},\ TGT(t_k) \notin \mathcal{S}(A) \\ st_A; & \mathit{if\ history\ state},\ TGT(t_k) \notin \mathcal{S}(A) \end{cases} \\
\\
\quad\quad \text{- - } \mathit{otherwise} \\
\quad\quad default : st_A; \\
\quad \}; \\
\\
\mathit{with}\ s \in \mathcal{S}(A_0)\ \mathit{such\ that}\ A \in \mathcal{A}(\rho(s)),\ \mathit{if}\ A \neq A_0.
\end{array}
$$

In practice, each sequential automaton will have two such `next` statements. One statement defines the state changes caused by null-triggered transitions, while the other statement specifies the state changes caused by triggered transitions. Naturally, each statement is placed in the correct `progress` branch block, as shown in Code Listing 3.9.

Clearly, Code Listing 3.9 illustrates that real parallellism is achieved for a concurrent state. Both because each concurrent region (e.g. `CoffeeIdle`) of a state is represented by a state variable, and because the `next` statements execute simultaneously.

Entering Transitions

Upon entering a sequential automaton, some cases are differentiated. The first option in the branch (Rule 3.9) attached to t_i is a *default entry* when an

Code Listing 3.9 State Changes

```
progress_auto & ~error : {
  next(st_Root) := case {
    t7     : Empty;
    t3     : Empty;
    t4     : On;
    default: st_Root;
  };
  next(st_Coffee) := case {
    t6     : CoffeeIdle;
    t7     : NotActive;
    t3     : NotActive;
    t4     : CoffeeIdle;
    default: st_Coffee;
  };
  next(st_Light) := case {
    t7     : NotActive;
    t3     : NotActive;
    t4     : LightOff;
    default: st_Light;
  };
  ...;
};
```

```
progress_trigger & ~error : {
  next(st_Root) := case {
    t19    : Off;
    t18    : On;
    default: st_Root;
  };

  next(st_Coffee) := case {
    t5     : CoffeeBusy;
    t19    : NotActive;
    t18    : CoffeeIdle;
    default: st_Coffee;
  };

  next(st_Light) := case {
    t17    : LightOff;
    t16    : LightOn;
    t19    : NotActive;
    t18    : LightOff;
    default: st_Light;
  };
  ...;
};
```

incoming transition terminates on the outside edge of the composite state. In this case, the default state is entered. A transition forces an *explicit entry* if the transition goes to a substate of the composite state, then that substate becomes active; the second option in the branch. This rule applies recursively if the transition terminates on a transitively nested substate; the last option in the branch. Finally, there is also a *history entry* meaning that the last visited substate becomes active again; this is not explicitly modeled.

As an example, consider transition $t18$ (Code Listing 3.9). Once the power of the machine is turned on, the concurrent state `On` is entered by default. This implies that the default states (e.g. `CoffeeIdle, LightOff`) of each concurrent region become active as well.

Hierarchical Conflicting Transitions

Suppose that at least two transitions t and t' conflict and that $t \sqsubset t'$ holds. This means that t' has the highest priority to execute. Obviously, in a `case`

block, the earlier it is placed, the earlier it is evaluated and taken. This guides the place of the enabledness conditions in such blocks. The definition of the target state is rather obvious.

The coffee vending machine does not contain any hierarchical conflicting transitions. Code Listing 3.9 only shows that the deeper a transition lies in the EHA, the earlier it is placed in the branch statement e.g. $t17$ is deeper in the hierarchy than $t18$. If they conflict (which cannot happen here) $t17$ is evaluated and taken first, as the UML semantics requires.

Equal Conflicting Transitions

Suppose again that at least two transitions t and t' conflict and have equal priority i.e. $t \sqsubset_= t'$ (Section 2.5). Then it is up to the machine (or to the model checker during verification) to select and fire one transition non-deterministically. CaSMV is forced to make such choices as defined in Rule 3.10.

Rule 3.10 (Equal Conflicting Transition Representation). *For each automaton $A \in F$ and for each $s \in \sigma_A$, let δ_{s_e} bet the set of all outgoing transitions of s triggered by the same event $e \in E$. The possible conflicts among these transitions (when enabled) are solved non-deterministically as follows:*

$$choose\_in\_s\_e : \{T \mid \exists t' \in \delta_{s_e} : \delta_{s_e} \ni \omega(T) \sqsubset_= t'\} \cup \{NotDefined\};$$
$$choose\_in\_s\_e := \begin{cases} \{\omega(T) \ ? \ T\}; \\ \{NotDefined \mid \textit{no transition is enabled}\}; \end{cases}$$

$\omega : \Delta \to \delta$ attaches to each symbol T a transition t.

As you can see, several `choose` functions are defined as enumerated macro variables. The type of such a variable is a set of symbols. These symbols are representations of transitions that leave the same state while triggered with the same event. Thus, there exists a chance that some of them, when enabled, are in conflict.

The value of these `choose` variables is defined using CaSMV *set expressions*. Such expressions are interpreted to represent a non-deterministic choice between the values in the set. A *set* is specified as a list of elements between curly brackets, and each element can be a *guarded expression* `c ? e`. In this case the value of `e` is included in the set if the condition `c` is true. Thus, each symbol T is included in the set if its corresponding transition t is enabled in the current configuration. However, if, in the current configuration, no transition enables, then the set will be a singleton set. Note that the `NotDefined` value is not included in the set, in case at least one enabledness condition is true.

History Mapping

If a transition causes the sequential state to be left in an undefined state, we set its next value to `NotActive`. Of course, if a history state is attached to the sequential automaton, its next value has to be set to its current value. At the moment the automaton is entered again, the last active visited substate becomes active as well, as required by the UML semantics.

3.7.5 Variable Valuation

Class attributes in a UML model can be assigned new values in actions performed by a transition. If an attribute value changes as a result of such actions, we use the transition macro as the enabling condition for the assignment. An attribute keeps its current value if a transition does not affect its value. The general rule is:

Rule 3.11 (Attribute Value Changings). *The behavioral update, with respect to the priority scheme, for class attributes is defined as follows:*

$$\begin{array}{l} case\ \{ \\ \quad t_i : \left\{ next(attr.attr_i) := \left\{ \begin{array}{ll} expression_i; & \text{if } attr.attr_i = expression_i \text{ belongs} \\ & \text{to the action list of } t_i \\ attr.attr_i; & \text{otherwise} \end{array} \right. \right. \\ \}; \end{array}$$

As before, there will be two such assignments since the transitions are divided into two disjunctive sets.

Code Listing 3.10 Attribute Value Changings

```
t14:= in_CoffeeReady & event_queue[0] = flow_coffeedone;

case {
  ...;
  t14: {
         next(attr.money) := money - 1;
         next(attr.cup)   := attr.cup;
         next(attr.coffee):= attr.coffee;
       };
  ...;
};
...;
```

3.7.6 Event Queue

Several examples will be used to explain how event variables are updated during verification, in order to respect the event processing rules.

While the machine executes a set of transitions, it is insensitive to newly arrived events ($\in E_i \cup E_f$), which accumulate in the queue attached to the statechart, until all transitions have finished their activities.

The generation of flow events caused by transition actions, which are carried out sequentially, defines the delivery order of the events. Concurrent transitions execute interleaved such that events can be generated in several orders. Combine this with the fact that external stimuli need to be generated at appropiate moments, then it cannot be a suprise that it is a rather difficult task to correctly define the transition relation of the event variables e.g. the trace of event delivery order must be reflected in the event queue.

Flow Events

As an example, Figure 3.6 shows two situations. The left part shows a transition `tr1` that, when executed, produces two flow events. Obviously, the trace of delivery order is (`flow_ev1`, `flow_ev2`). The right part shows two concurrent transitions (`tr2` and `tr3`). When they both are performed during a step, the delivery order is either (`flow_ev3`, `flow_ev4`, `flow_ev5`) or (`flow_ev5`, `flow_ev3`, `flow_ev4`) due to the interleaved model of computation inside concurrent states.

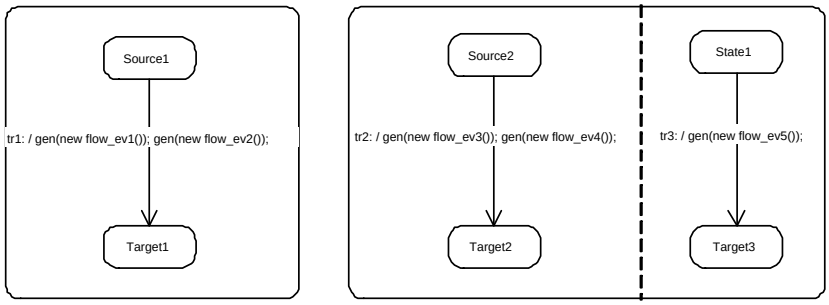


Figure 3.6: Trace of Flow Events Delivery Order

Code Listing 3.11 shows how both orders are reflected into $\mathcal{M}$. As previously mentioned, the `event_tail` is used to place newly generated events into the queue with respect to both the delivery order and the FIFO order. Additionally, such updatings have to conform to the single assignment rule. This is achieved by using a `for loop` and by carefully specifying the branch

conditions; it is impossible that two branch conditions can be taken for the same index i. The latter conditions are defined using transition macros as enabling conditions, again with respect to priorities.

Code Listing 3.11 Event Queue Update with Flow Delivery Orders

```
for (i = 0; i < SIZE; i = i + 1) {
case {
 tr1 & event_tail = i       : next(event_queue[i]):= flow_ev1;
 tr1 & (event_tail + 1) = i : next(event_queue[i]):= flow_ev2;
 tr2 & ~tr3 & event_tail = i: next(event_queue[i]):= flow_ev3;
 tr2 & ~tr3 & (event_tail + 1) = i:
             next(event_queue[i]):= flow_ev4;
 ~tr2 & tr3 & event_tail = i: next(event_queue[i]):= flow_ev5;
 tr2 & tr3 & event_tail = i:
             next(event_queue[i]):= {flow_ev3, flow_ev5};
 tr2 & tr3 & (event_tail + 1) = i:
     next(event_queue[i]):=
       case {
        i > 0 & next(event_queue[i-1]) = flow_ev3: flow_ev4;
        default                                  : flow_ev3;
       };
 tr2 & tr3 & (event_tail + 2) = i :
     next(event_queue[i]):=
       case {
        i > 0 & next(event_queue[i-1]) = flow_ev4: flow_ev5;
        default                                  : flow_ev4;
       };
 default : next(event_queue[i]) := event_queue[i];
};
};
```

If the action list of a transition (e.g. tr1) generates multiple events, then multiple branch statements are inserted into $\mathcal{M}$. Concurrent transitions, like tr2 and tr3 are treated in an extraordinary way since $\mathcal{M}$ must contain every possible behavioral combination. The side effect is the highly complex way to achieve the correct trace of event delivery order.

Updating the variables event_tail and event_overflow is much easier (Code Listing 3.12) but we must be aware of queue overflows. If this happens the model checker is forced to infinitely reside in a special error state: StateMachineError.

Code Listing 3.12 Event Tail/Overflow Update

```
case {
  tr1 | (tr2 & ~tr3) :
              if ((event_tail + 2) >= SIZE) {
                 next(event_tail)    := event_tail;
                 next(event_overflow):= 1;
              } else {
                  next(tail)          := tail + 2;
                  next(event_overflow):= event_overflow;
                };
  ~tr2 & tr3 : if ((event_tail + 1) >= SIZE) {
                  next(event_tail)    := event_tail;
                  next(event_overflow):= 1;
               } else {
                   next(event_tail)    := event_tail + 1;
                   next(event_overflow):= event_overflow;
                 };
  tr2 & tr3 : if ((event_tail + 3) >= SIZE) {
                next(event_tail)    := event_tail;
                next(event_overflow):= 1;
              } else {
                  next(event_tail)    := event_tail + 2;
                  next(event_overflow):= event_overflow;
                };
};
```

Interaction Events

External stimuli are the events used by the statechart to communicate with the outside world. They have the property that they might be available in the environment. But this is not a necessary prerequisite, since the behavior of the environment is unpredictable. For example, a user inserts a coin in the coffee vending machine or not.

Since the statechart repudiates all responsibility to create these stimuli and does not know when they might even occur, the transition behavior has to simulate the occurrence of those events using the event generator. Stated otherwise, the transition behavior has to play the role of the outside world whenever it might be appropriate.

The coffee vending machine moves from `Off` to `On` when a `inter_poweron` event occurs. When the machine is in `StandBy` coins can be inserted or returned. But the machine can change state from `On` to `Off` whenever the

user wants it too.

External stimuli accumulate in the queue, as shown in Code Listing 3.13. Now, a macro variable, e.g. `poweroff`, is used to avoid that the machine is immediately turned off, when it resides in the `On` state. Otherwise, the machine never has the chance to prepare some coffee. E.g. at the moment `t12` takes care of some evolution, the event `flow_cupstart` is added at the end of the queue. At the same time, it is possible that somebody turns the machine off. Following the RTC step semantics, the machine reaches state `Off` after `t12` has finished execution. Moreover, because `inter_poweroff` is not modeled as an interrupt, it accumulates separately in the queue. Note, that the guard on the position in the queue has slightly changed; as will be clarified later on.

Code Listing 3.13 Event Queue Update with External Stimuli

```
t12:= in_StandBy & event_queue[0] = inter_button & (money > 0);
t13:= in_CupReady & event_queue[0] = flow_cupdone;

poweroff: {InterPowerOff, NotDefined};
poweroff:= {InterPowerOff, NotDefined};

for(i=0; i<SIZE; i=i+1) {
...;
case {
  ...;
  t12 & event_tail = (i+1):
      next(event_queue[i]):= flow_cupstart;
  t12 & (event_tail+1) = (i+1) & poweroff=InterPowerOff:
      next(event_queue[i]):= inter_poweroff;

  t13 & event_tail = (i+1) & poweroff=NotDefined:
      next(event_queue[i]):= flow_coffeestart;
  t13 & (event_tail+1) = (i+1) & poweroff=InterPowerOff:
      next(event_queue[i]):= inter_poweroff;

  t18 & event_tail = (i+1): next(event_queue[i]):=
      {inter_coin, inter_return, inter_button, inter_poweroff};
  ...;
};
};
```

Influence of Progress Phases

So far, we have updated the event variables without explicitly considering the progress blocks. What happens with their behavioral updates when placed correctly in both phases of the step relation? Basically, it is as good as the same, but there are some remarkable differences.

A first notable difference is the manipulation of the queue (Code Listing 3.14). Obviously, if the dispatcher decides to fire null-triggered transitions, all new events are inserted at the tail (denoted by `event_tail = i`) of the queue. However, when an event is dispatched, it is also shifted out off the queue. As a direct consequence, the first position to insert a new event is marked by the condition `event_tail = (i+1)`. This is exactly the position before the one the tail points to, since elements are moved from the back to the front of the queue.

Code Listing 3.14 Event Queue Update with Tail Consideration

```
progress_auto & ~error : {
for(i = 0; i<SIZE; i=i+1) {
case {
 -- update queue
 -- position:
 --    (event_tail)  = i,
 --    (event_tail+1)= i,
 --    ...
 -- next(event_queue[i]):=...;
};
};
...;
};
```

```
progress_trigger & ~error : {
for(i = 0; i<SIZE; i=i+1) {
-- shift head out
if ( ((i + 1)<event_tail)
     &(i + 1)<SIZE)
  next(event_queue[i]):=
          event_queue[i+1];
else {
    if( ~event_tail=0) {
    case {
    -- update the non-empty queue
    -- position:
    --    (event_tail)  = (i+1),
    --    (event_tail+1)= (i+1),
    --    ...
    -- next(event_queue[i]):=...;
    };
    } else {
       -- same as in progress_auto
    };
  };
};
};
```

Another worthwhile difference is the way the tail variable is updated. Either the pointer is moved as much places as the total number of produced events,

or it is moved one position less. Only this way, at any time, `event_tail` will point to the first empty queue position.

3.7.7 ... inside Stuttering Phases

The system may stop executing when a exception (e.g. queue overflow, race conditions) has occurred. These mistakes forces the model checker to infinitely leave the machine in a special error state.

Rule 3.12 (Stuttering in an Exception Configuration). *A system that remains infinitely often in an exception configuration, is modeled as follows:*

$$
\begin{array}{lll}
\textit{error: } \{ & & \\
\quad \textit{next}(\textit{st\_}A_0) & := \textit{StateMachineError}; & \\
\quad \textit{next}(\textit{st\_}A) & := \textit{NotActive}; & \forall A \in F \setminus A_0 \\
\quad \textit{next}(\textit{attr.attr}_i) & := \textit{attr.attr}_i; & \forall \textit{attr}_i \in \textit{attr} \\
\quad \textit{next}(\textit{event\_queue}) & := \textit{event\_queue}; & \\
\quad \textit{next}(\textit{event\_tail}) & := \textit{event\_tail}; & \\
\quad \textit{next}(\textit{event\_overflow}) & := \textit{event\_overflow}; & \\
\}; & &
\end{array}
$$

If there does not exist an automaton $A \in F$ to which the execution of transitions can be delegated, then H has to stutter, as enforced by the stuttering rule 2.15. Such a lack of progress is modeled in CaSMV using Rule 3.13.

Rule 3.13 (Stuttering due to Lack of Progress). *The stuttering rule for each EHA is modeled as follows:*

$$
\begin{array}{lll}
\textit{default: } \{ & & \\
\quad \textit{next}(\textit{st\_}A) & := \textit{st\_}A; & \forall A \in F \\
\quad \textit{next}(\textit{attr.attr}_i) & := \textit{attr.attr}_i; & \forall \textit{attr}_i \in \textit{attr} \\
\quad \textit{next}(\textit{event\_queue}) & := \textit{event\_queue}; & \\
\quad \textit{next}(\textit{event\_tail}) & := \textit{event\_tail}; & \\
\quad \textit{next}(\textit{event\_overflow}) & := \textit{event\_overflow}; & \\
\}; & &
\end{array}
$$

3.8 Template Embedded Models

A template embedded model is a model that allows us to convert arbitrary embedded statecharts to the modal logic of CaSMV, covering all behavioral model elements as defined in the UML specification, in

particular, hierarchy, sequentialism, parallelism, non-determinism, priorities, and run-to-completion step semantics. After augmenting the model with expectation properties, the CaSMV model checker is subsequently used to verify behavioral design.

To show how an embedded model looks like in CaSMV, it is sufficient to cover the basic structure of each model in one module (Code Listings 3.15 to 3.17); i.e. the `main module`.

Code Listing 3.15 Template Embedded Models

```
/* Define the data types
     1. Define size of the event queue
        E.g. #define SIZE 3
     2. Type definitions of the class attributes
        (Section 3.5.2)
**/

main module () {

  /* Variables Declaration
       1. ... of the automata
          (Section 3.5.1)
       2. ... of the class attributes
          (Section 3.5.2)
       3. ... of the event variables
          (Section 3.5.3)
  **/

  /* Macro Variables Declaration/Instantiation
       1. ... of the in_substate conditions
          (Section 3.7.2)
       2. ... of the enabling and choose conditions
          (Section 3.7.3)
       3. ... of the exception condition
          E.g. error:= queue_overflow;
       4. ... of interaction conditions (event generator)
          (Section 3.7.6)
  **/
```

The template, we have introduced here, makes way for the automatic translation of UML statecharts[4]. In particular, we will use the standardized

[4]See also Section 3.2.

Code Listing 3.16 Template Embedded Models Continued

```
/* Initialization
     1. ... of the automata
        (Section 3.6.1)
     2. ... of the class attributes
        (Section 3.6.2)
     3. ... of the event variables
        (Section 3.6.3)
**/
```

translation of UML in XML to fill out the parameters of our template. With such an automatic translation available, UML would not only be the standard for embedded system development, but also the portal to formal verification. Hence UML would become a more powerful and interesting tool, particularly for engineers and programmers.

3.9 Assumptions Made

For reasons to be complete, we provide the assumptions we have made during the construction of an embedded Kripke model.

3.9.1 Actions attached to States

An action in a transition indicates that an atomic computation is performed when the transition fires. Actions can be attached to states as entry/exit actions or associated with internal transitions as well. Entry/Exit actions can be substituted with transition actions without changing the semantics of a statechart [47]. After replacing them, internal transitions can be modeled easily as self loops, again without affecting the semantics. During the firing of a transition, the execution of events and actions happens in the following order: event, exit action of the source state, action sequence attached to the transition, and entry action of the target state finally.

So far, we know that a Kripke model captures the basic concepts of the state of a system and the actions (transitions) that modify the state. Based on this definition, the states of an EHA ($\in \mathcal{S}(A_0)$) are not assumed to have any action during the model construction. Instead, entry actions are moved to the incoming transitions and are the last ones in the sequence of actions executed by the incoming transitions; exit actions are moved to the outgoing transitions and are the first ones in the sequence of actions executed by

Code Listing 3.17 Template Embedded Models Continued

```
    /* Transition Relation

       case {
         progress_auto & ~error: {
            -- updates caused by null-triggered transitions
            -- (Sections 3.7.4 to 3.7.6)
         };
         progress_trigger & ~error: {
            -- updates caused by triggered transitions
            -- Section 3.7.7
         };
         error: {
            -- exception configuration
            -- Section 3.7.7
         };
         default: {
            -- lack of progress
            -- Section 3.7.7
         };
       };

    **/

    /* Specification
          expectation phrases in LTL/CTL
    **/
  }
```

these transitions; internal transitions are converted into self-loops, of course without entry or exit actions.

3.9.2 Actions attached to Transitions

Our model checker has to analyze transition actions according to their written order, since the actions order of execution is highly relevant. Additionally, it is a violation against the single assignment rule of CaSMV when the transition's action sequence (including entry/exit actions) defines at least one class attribute twice. If a transition violates this rule, then the transition is split up into several unique transitions without a trigger and basic states, in such a way that each transition respects the single assignment rule (see

figure 3.7).

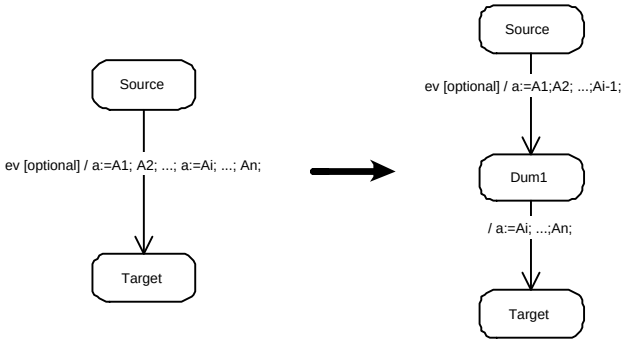


Figure 3.7: Splitting up Transitions

The statechart without injections (sc) and the one with injections (sc_i) are two ways of specifying the same embedded system. But a model for sc_i will be constructed. sc_i differs from sc in the sense that at least one transition of sc is replaced by a complete range of newly created transitions and states.

Not surprisingly, the equivalence between sc and sc_i has to be retained during verification. Two systems are equivalent if they have the same semantics meaning that the same information can be derived from their behavior.

Suppose the embedded system is specified by a single sequential automaton. Due to the absence of concurrent states, the semantically equivalence between sc and sc_i is proven easily since the bifurcations do not change the sequential execution of statements.

The presence of concurrent states makes things more complicated. Orthogonal regions represent independent states that may concurrently be active with other states. The operational semantics is based on the premise that a single run-to-completion step applies to the entire state machine and includes the parallel steps taken by the concurrent regions in the current active configuration. When we examine the regions as cooperating entities, can we still talk about equivalence? In fact this is not always guaranteed. E.g. the splitting up of transitions may lead to a change in the trace delivery order of newly generated events.

To guarantee the equivalence between sc and sc_i, such aspects have to be considered, since they influence the information that can be derived from its behavior: another event delivery order implies behaviors that are not present in the original statechart. Thus, a manipulation of the operational semantics is the only way out.

Let's illustrate our reasoning by examining Figure 3.7. Suppose that in *sc* transition `t1` and transition `t2` execute concurrently in the same RTC-step. Following the operational semantics, the active configuration `AB` evolves into configuration `A'B'` while dispatching a single event. The same must be true in the model sc_i: configuration `A'B'` must be reached while dispatching a single event. What is the bottleneck? The current structure of the templates is such that in each time step during model checking, real parallelism is achieved in case concurrent states belong to an active configuration. If we keep this way of working, transition `t1_1` and transition `t2` execute concurrently and transition `t1_2` might be fired with some other null-triggered transition leaving state `B'`. Obviously, if this happens, the execution order and the event delivery order of both statecharts will possibly be different. This can be avoided by interrupting the run-to-completion steps at obligatory moments.

A Non-Trivial Solution

Guaranteeing the equivalence poses a real challenge in the construction of $\mathcal{M}$. Since orthogonal states run independently of each other, race conditions (e.g. two regions changing the value of the same variable, one region using the value of a variable that another one is changing) may not occur. This gives us everything we need to correctly implement equivalence, using an interrupt. If a concurrent state of sc_i contains states that are not present in the corresponding concurrent state of *sc*, then a *private queue* is attached to it. At the moment we detect — using a lookahead mechanism — that an active concurrent configuration evolves to a concurrent configuration containing one or more newly introduced states and transitions, the interrupt is called.

The interrupt has several responsibilities. Firstly, it is capable to achieve real parallelism in the same manner the operational semantics achieves it. In general the following steps are taken:

- Achieve real parallelism between those transitions leaving states from the active concurrent configuration. This results in reaching a configuration containing newly introduced states in some of the regions. In our example the interrupt executes transition `t1_1` and `t2` concurrently.
- Achieve real parallelism between those transition leaving newly introduced states. In our example transition `t1_2` will be taken by the interrupt. This step can be repeated several times.

Secondly, although real parallelism is achieved, the event delivery order will be respected since the interrupt is obliged to update the private queues of

the regions. The general queues used to implement the run-to completion semantics are not affected! Only the first step of the interrupt is allowed to make an exception. The first step will remove the event from the corresponding general queue. Last, a lookahead mechanism is used again to detect wether a real configuration (containing only states present in the original statechart) can be reached. If so, the private queues are merged (several orders) to update the global queues. Now, the interrupt has finished working and a new event can be dispatched now.

Undoubtedly, modeling such an interrupt this way is not a trivial case. Therefore, we assume that the action sequence of each transition changes class attributes only once. On the other hand, it is debatable whether multiple assignments will be present at design stage.

3.9.3 Firing Multiple Transitions

The semantics of UML statecharts allows for the possibility of non-determinism in state transitions. Multiple transitions, triggered by the same event, may be enabled for firing from the same source state at the same time. At each step, only one of the transitions leaving the same state is allowed to fire. Nonetheless, the simultaneous firing of multiple transitions is allowed only, if these transitions are in separate orthogonal components. Unfortunately, there are some situations where the UML semantics does not provide a clear solution.

Suppose that `t1` and `t2` (Figure 3.8) fire simultaneously, and the corresponding target states are reached properly. However, either `Target1` or `Target2` is made active since they belong to a sequential automaton. But which one exactly, is not defined in the UML semantics. We believe that such a situation is rather akward and therefore we omit it.

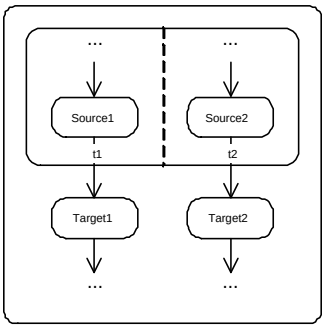


Figure 3.8: Problem when Firing Multiple Transitions

3.10 Conclusions and Related Work

The behavior of embedded systems is graphically specified using standalone UML statechart diagrams. The behavioral properties of these systems are verified using the model checking technique, which is integrated in an automated verification tool. This chapter has illustrated how an embedded standalone statechart can be translated into a CaSMV Kripke model, which is a necessity to exhaustively verify the UML design model. Do not forget that we aim at a correct design before actually implementing the software. The chapter has also illustrated how the verification methodology is capable to handle open models. This allows software developers to verify open designs where the environment is not (yet) completely specified. It may be of no surprise that our methodology can be integrated in an iterative model based design process.

We will now investigate existing approaches to the formal verification of statecharts using model checking and we will point out the main disadvantages of the other approaches (see also [15]). All the methodologies have one thing in common: they translate the hierarchical structure of the statechart model to the input language of a particular model checker. So, the black box of the different approaches will be more or less the same; only the details differ.

The paper by Latella et al. [72] proposes a translation from a subset of UML statecharts into Promela; the input language of the model checker SPIN [54, 55]. The subset considered covers the aspects related to concurrency and state hierarchy. In contrast with our transformational approach, variables and history states are not covered but can be handled conceptually. Moreover, the operational semantics is heavily restricted to triggered transitions only, meaning that instable configurations are never encountered during verification. Obviously, this dramatically simplifies the semantics of the transition relation. Another disadvantage of Latella's approach is that the translation does not consider multiple statecharts[5] which communicate with each other through event queues. However, the authors, having doubts about the methodological soundness of such an approach, claim that the translation can be easily adapted to handle communicating statecharts as well. The paper by Darvas et al. [35] extends the approach of [72], in such a way that multiple statecharts can be handled. Unfortunately, they do not provide the semantics that they have followed, nor give some transformational details.

[5] The verification of multiple statecharts is addressed in the next chapter.

The work by Lilius and Paltor [99] discusses a formalisation of UML state machines for the translation to Promela as part of their verification tool vUML [77]. Compared with most other works, the subset of UML statecharts considered is bigger. Although, the tool vUML performs verification automatically by invoking SPIN [55, 54], the performed verification is restricted to essentially deadlock detection, or overrunning a message queue. Compared with our work (addressed in this chapter and the next one), their operational semantics does not consider the case where objects share the same execution thread. They also do not make a distinction between different types of events that play a major role in the construction of a Kripke model for communicating statecharts. Again, this is a simplification of the STEP semantics and the way it has to be faithfully translated into the Promela language (= modeling language of SPIN).

Some work has also been done in translating statechart diagrams to variants of the model checker SMV. The papers [23, 27] propose a very incomplete STEP semantics e.g. the trace of event delivery order is not respected at all. On the other hand, the paper by Beato et al. [10] presents a tool which enables the active UML behavior of a system to be verified in a complete automatic way. The proposed model has some similarities with our model but again the trace of event delivery order is neglected. Also, they did not have mentioned how the priority scheme is guaranteed. One of its advantages is the incorporation of an assistant that acts as a user guide for writing properties in temporal logic.

The above overview has clearly shown that our semantics is quite complete. The next chapter provides the semantics and the translation process for (a)synchronously communication statecharts. An illustration of the way the verification process can be integrated in the UML design can be found in Chapter 9.

CHAPTER 4

The Model of Communicating Statecharts

I am a computer, dumber than any human
and smarter than any administrator.
Computer One Liners.

Embedded systems are composed, to a large extent, of event components which must react continuously to external stimuli from their environment. The typical task of such systems is to control the cooperation of several active processes in the system's environment. Embedded applications will be typically characterized by many components in such a way that they have to simultaneously control several active external processes. Designers of embedded systems use statecharts extensively for the behavioral description of these components.

When systems are modeled as a set of components, issues of communication or synchronization necessitates a thorough verification of the overall system behavior. Most problems of embedded systems arise from the coordination of interactions, which may result in deadlocks or other unwanted (hazardous) states. As the complexity of the systems increases, the manual verification of these properties becomes more and more error prone. Automatic support is required to help the designer in this task.

This chapter presents an automated verification process of embedded systems design using a collection of UML statecharts. The method extends the methodology discussed in Chapter 3. Again, the verification is performed by an automatic model transformation from the UML models to CaSMV, still through extended hierarchical automata. The transformation preserves the behavioral semantics of the system, which is in turn an extension of the semantics presented in Chapter 2.

4.1 Motivating Example

We introduce the *Automated Rail Car System* described in [50], which will be used throughout this chapter as an example to explain and illustrate the main ideas. The Automated Rail Car System is a model of autonomous rail-bound cars which transport passengers between terminals and which adhere to a simple arrival and departure protocol to allocate and deallocate platforms inside the terminal.

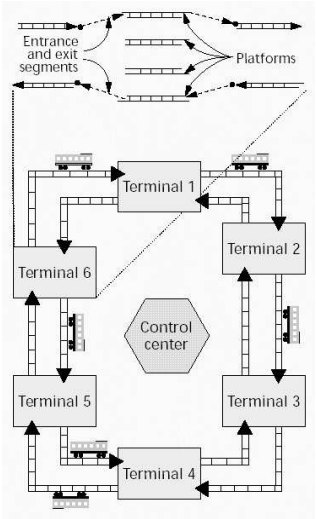


Figure 4.1: A Rail Car System

4.2 Object Communication

The state machine view describes the dynamic behavior of objects over time by modeling the life cycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. Events represent the kind of changes that an object can detect such as the receipt of explicit signals or calls sent from one object to another [38, 96, 104, 105]. Both types of events can have a name and carry a list of parameters.

Asynchronous with Signals A *signal* is an event explicitly intended as a communication vehicle between two objects; the reception of a signal is

an event for the receiving object. The sending object explicitly creates and initializes a signal instance and sends it to a single object or to a set of objects. Signals embody *asynchronous* one-way communication. The sender does not wait for the receiver to deal with the signal but independently continues with its own work. To model two-way communication, multiple signals can be used, at least one in each direction.

Synchronous with Calls A *call* is a *synchronous* communication (waits for a response) that represents the dispatching of an object operation. The parameters of the operation are the parameters of the call event. We choose to have a call event whenever an object sends out a message, the control is passed to the receiver, the receiver changes its state and returns the control to the sender. Basically, this means that the caller waits for the callee before continuing its own execution.

Example 4.1. *Suppose the objects* `car:Car` *and* `hnd:CarHandler` *are part of the Automated Rail Car System from [50]. These objects use statecharts to describe behavior that can be different under different circumstances, as partially shown in Figures 4.2-4.3 respectively. The synchronous communication between* `car` *and* `hnd` *is achieved by sending out a call event like* `c_departAck`*.*

4.3 Object Concept

Model checking the communication between objects will not succeed without thoroughly examining the static structure of the embedded system. This structure tells us whether instances of classes are either *active* or *passive* instances. The UML allows both active and passive objects to have state machines without any constraints.

Active Objects When it comes to concurrency, the UML offers the concept of active objects. An active class indicates that, when instantiated, it will control its own execution. Rather than being invoked or activated by other objects, it can operate standalone, and define its own thread of control (behavior) [110]. Here, the *thread of control* represents an abstract notion of control and not an operating system thread.

As only one message may be treated at a time (step semantics), there will have to exist some mechanism for queuing the messages. Whether this queuing mechanism exists or how it acts, is not said in [96]. Following the Rhapsody semantics [48], each active object is a one threaded object

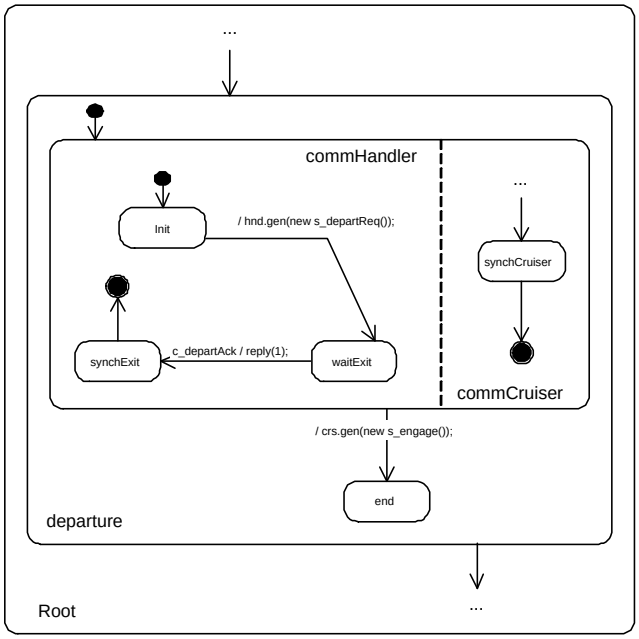


Figure 4.2: Statechart Diagram for a Car

that performs event management using its own event queue and queuing mechanism [48].

Passive Objects In contrast, a passive object, by not having its own thread of control, does not have a scope for executing the state machine. As a consequence the state machine will not run continuously as in the case of active objects, instead it will only be able to be executed while the passive object can run on some other object's thread of control. The behavioral specification of passive objects raises some questions [44]:

- Where are the messages directed towards a passive object stored?
- If there is a queue, how is it managed?
- On which execution thread does it execute?

These problems will be solved later on.

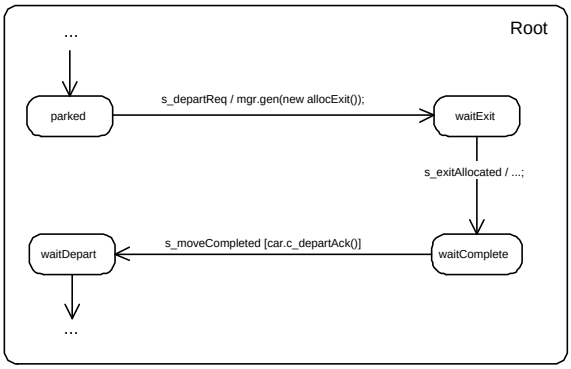


Figure 4.3: Statechart Diagram for a CarHandler

4.3.1 Multi-Threading

Due the presence of active objects, embedded applications become multi-threaded applications. Each thread performs steps in parallel to the other threads. In practice, an interleaved model of computation is often used. Obviously, this makes the definition and the behavior far more complicated than in the case of a standalone statechart.

Object/Thread Relationship Which objects belong to which threads? When a class is defined as an active class, the instances of this class will have their own thread of control. Objects can also be related to threads through composition. This means that components of a composite class will run on the thread of the latter one, unless they are instances of an active class, in which case they have their own thread of control. Instances of classes that are not designated as active classes, run on the default thread of control which is used by the main program [48].

Summarized, each thread of control contains at most one active object, and allows to include an arbitrary number of passive objects in the group. Each active object, as owner of the thread of control, thus has its own event queue, whereas passive objects share their event queue with some active object. Obviously, all event handling is delegated to the shared event queue. When the event has reached the top of the queue, the event is taken from the queue and dispatched to the destination object.

Example 4.2. *Figure 4.4 (inspired by [34]) illustrates the multi-threading concept using the Automated Rail Car System from [50]. It is a snapshot of a model part and shows active objects* `car` *and* `term:Terminal`*. Each* `car`

comes equipped with a cruise controller (passive object **crs***) for maintaining speed.* **hnd** *on the other hand handles the transactions between a car and a terminal. Clearly, active objects* **car** *and* **term** *designate their threads of control to which their components belong.*

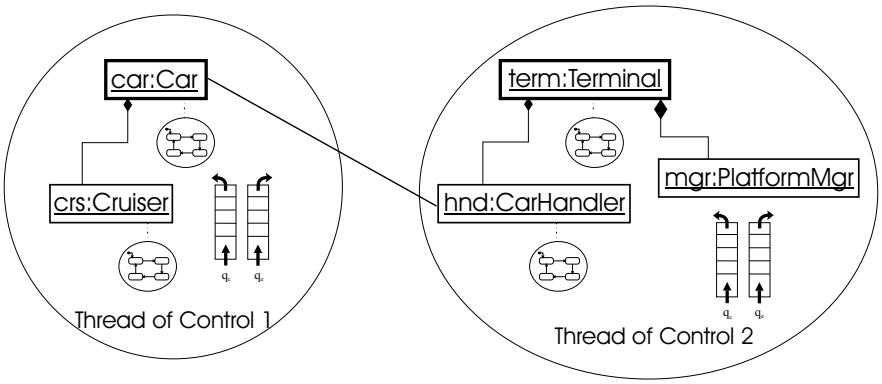


Figure 4.4: Threads of Control

Communication between Threads Statecharts of objects can deal with asynchronous communication using events and with synchronous communication using calls. In the multi-threaded case, objects receive messages from different objects. To avoid that events get lost, the queuing mechanism will be extremely important.

The case of asynchronous communication is simple. The generated events are put in the event queue of the thread to which the receiving object belongs, and dispatched to the statechart at the appropriate moments [48].

In the case of synchronous communication, the thread to which the sending object belongs is blocked until the receiving statechart completes its response to the call event, which accumulates in the call queue[1] of the thread containing the callee [48]. Due to the locking mechanism, situations of deadlocks and starvation are possible.

Example 4.3. *Sending a call event (*`c_departAck`*) from* **hnd** *to* **car** *in fact enters the event into the call queue q_c of the first thread of control. Sending a signal event (*`s_departReq`*) from* **car** *to* **hnd** *causes the event to be entered in the event queue q_e of the second thread of control. Communication between*

[1]The reason for having two queues for both event types is explained in Section 4.4.

`car` and `crs` (Figure 4.5) based on events ($\in E$), causes these events to be entered in the queues of the thread to which they belong.

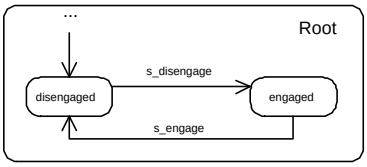


Figure 4.5: Statechart Diagram for a Cruiser

4.4 Operational Semantics Extended

The step algorithm for a multi-threaded embedded system consists of performing the step cycle described in Chapter 2 for each thread. Unfortunately, there are several complications in the semantics relative to those of a standalone statechart, particularly concerning the dispatching mechanism.

Having a clear understanding of the semantics is important to build a concise model used by the verification process. As before, the semantics will be defined in terms of extended hierarchical automata. But now, the set of events $E = E_i \cup E_f \cup E_s \cup \varepsilon$ is the union of the set of interaction events, flow events, signal events, and the null-event respectively. Call events ($\in Ec$) do not belong to E; they will be treated and dispatched separately.

4.4.1 Threads Containing Exactly One Object

The *asynchronous communication* between active objects only, easily integrates in the RTC-step semantics. Basically, generated event instances are inserted into the thread's event queue q_e and hold there until the instances are dispatched. More precisely, signal events are treated for dispatching like any other event; they will be dispatched after they have reached the head of the queue.

Active objects execute based on an interleaved model of computation. At any time, exactly one active object will execute. Knowing this, we safely conclude that the RTC-step semantics, belonging to threads (covering exactly one object) is exactly the same as the semantics attached to a standalone statechart (see Chapter 2). To each active object, an event (either an interaction, a flow, a signal event or the null-event) is dispatched in a stable configuration and its statechart is evaluated until a stable state configuration

is properly reached.

On the other hand, the *synchronous communication* between active objects complicates the RTC-step semantics. As discussed previously, synchronous communication is achieved through call events. Call events are special events in the sense that they represent a synchronous invocation of a specific operation i.e. they are created by method call actions. When a method call is executed as an action, the whole transition step is completed only when the invoked objects (= callees) complete their own run-to-completion steps. This means that the sender of a call event is blocked until the callee has consumed the event.

The blocking mechanism necessitates us to give a higher priority to synchronous messages than to asynchronous ones. In the dispatching process, we first try to dispatch a call event and if that is not possible, then we try to dispatch an event (see Definition 4.1). The basis for dispatching call events is, again, a FIFO queue q_c (shared by the objects belonging to the same thread of control); call events are served to the statechart in the order they were put in (from the oldest to the youngest). If a call event arrives when the state machine is not in an appropriate state to handle the event, the event is discarded, conforming the general RTC-step semantics. Strictly speaking, from the caller's point of view this means that the call is completed.

Definition 4.1 (Dispatcher). *For a configuration $\mathcal{C}$, dispatch($\mathcal{C}$) returns the event ($\in E \cup Ec$) to be dispatched in $\mathcal{C}$:*

$$dispatch(\mathcal{C}) = \begin{cases} \varepsilon & \text{if } MaxET(\mathcal{C})_{\varepsilon} \neq \emptyset \\ q_c[0] & \text{if } MaxET(\mathcal{C})_{q_c[0]} \neq \emptyset \\ q_e[0] & \text{if } MaxET(\mathcal{C})_{q_e[0]} \neq \emptyset \\ nil & otherwise \end{cases}$$

Having two separated queues to store the events changes the run-to-completion step semantics (Chapter 2) in some ways. Still, a step is a sequence of transitions between two stable configurations. As before, this means that once an event (signal, call, interaction, flow, ϵ) is dispatched, the system evolves on its own until no more transitions can be taken. Then, the dispatcher has to be called once again. Definition 4.2 shows how another progress rule is injected in Definition 2.16 in such a way that a higher priority is given to calls than to signals. Obviously, each configuration of $\mathcal{K}_t$ contains two queues: q_c and q_e.

Definition 4.2 ($\overset{STEP}{\rightarrow}$). *The transition relation of a Kripke structure $\mathcal{K}_t$ is*

defined as follows:

$$\overset{STEP}{\rightarrow} = \begin{cases} \overset{prog}{\longrightarrow}_{\varepsilon} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=\varepsilon} \neq \emptyset \\ \overset{prog}{\longrightarrow}_{q_c[0]} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=q_c[0]} \neq \emptyset \\ \overset{prog}{\longrightarrow}_{q_e[0]} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=q_e[0]} \neq \emptyset \\ \overset{stut}{\longrightarrow} & otherwise \end{cases}$$

with $\mathcal{K}_t$ the Kripke structure of an active object (a thread).

It is sufficient to build the Kripke structure $\mathcal{K}_t$ for each active object (thread) separately. They form the basis for the Kripke structure $\mathcal{K}$, which serves as the Kripke structure of communicating statecharts. Obviously, $\mathcal{K}$ operates interleaved which means that only one state component changes value in a given transition. The choice of which component changes value is non-deterministic. In general, for models of communicating threads (parallel processes) whose actions interleave arbitrarily, the transition relation is disjunctive. Thus, we simply avoid to explicitly compute the transition relation of the communicating statecharts.

4.4.2 Threads Containing Multiple Objects

Sometimes several objects execute on the same thread of control (Section 4.3.1) while sharing the same event queue. These objects possibly communicate (a)synchronously with each other or with objects belonging to other threads of control. The problem is now to describe the step cycle for each thread carefully.

At the moment a stable state configuration is reached, the dispatcher is called. Due to the presence of multiple objects, the dispatcher selects an event from one of the queues (either $q_c[0]$ or $q_e[0]$) and sends it to the statechart of the object that responds to the message; which is of course unique. Thereafter, the statechart of the target object is repeatedly evaluated until a stable configuration is reached. Only then, the event is fully consumed and the next event can be dispatched, but possibly another object's statechart responds to the latter event.

However, before serving a new event, the dispatcher needs to verify whether there are other objects that became unstable due to the progress made in the last evaluated statechart. Following the step semantics, these objects must become stable before it is allowed to dispatch a new event. The whole step cycle is summarized in Definition 4.3.

Definition 4.3 ($\overset{STEP}{\rightarrow}$). *Let 'last' be the statechart to which an event is last dispatched. Let 'auto' be one of the objects that became unstable due to the*

progress made in 'last'. Then, the transition relation of a Kripke structure $\mathcal{K}_t$ is defined as follows:

$$\overset{STEP}{\rightarrow} = \begin{cases} (\overset{prog}{\longrightarrow}_{\varepsilon})_{last} & \text{if } (MaxET(\mathcal{C})_{dispatch(\mathcal{C})=\varepsilon})_{last} \neq \emptyset \\ (\overset{prog}{\longrightarrow}_{\varepsilon})_{auto} & \text{if } (MaxET(\mathcal{C})_{dispatch(\mathcal{C})=\varepsilon})_{auto} \neq \emptyset \\ \overset{prog}{\longrightarrow}_{q_c[0]} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=q_c[0]} \neq \emptyset \\ \overset{prog}{\longrightarrow}_{q_e[0]} & \text{if } MaxET(\mathcal{C})_{dispatch(\mathcal{C})=q_e[0]} \neq \emptyset \\ \overset{stut}{\longrightarrow} & otherwise \end{cases}$$

with $\mathcal{K}_t$ the Kripke structure of an active object (a thread).

4.5 Model Checking with CaSMV Revisited

CaSMV allows us to describe an embedded system as a set of finite state transition systems, each of which operates interleaved to each other. Therefore, it will be sufficient to map each thread of control to a *module* of the language.

A *process* is an instance of a module which is introduced by the keyword `process`. At any given time, the process is either running or not running. An assignment to the `next` value of a variable only applies if the process is running. If not, the value remains the same in the next time step. The interval between times that a process runs is arbitrary i.e. non-deterministic. This means that the processes operate *interleaved* to each other.

A `fairness constraint` must be used in order to force a given process to execute infinitely often. Each process has a special variable called *running* which is *one* if and only if that process is currently executing.

4.5.1 Introductory Example

Code Listing 4.1 (taken from [87]) shows us an interleaving model of execution for a ring of three asynchronous inverting gates. The Kripke model is defined by instantiating three times the module type `inverter` in the module `main`, with the names `gate1`, `gate2` and `gate3` respectively.

The `inverter` module, which in fact defines a sub-Kripke model in turn, has one formal parameter `inp`. In the instance `gate1`, `inp` is given the value of the expression `gate3.outp`. This expression is evaluated in the context of the main module. However, an expression of the form `a.b` denotes component `b` of module `a`, just as if the module `a` were a data structure in a standard programming language.
At most one of the three processes is allowed to run at any given time (interleaving semantics). The specification of the program states that the

Code Listing 4.1 Another Example CaSMV Program

```
module main() {
  gate1: process inverter(gate3.outp);
  gate2: process inverter(gate1.outp);
  gate3: process inverter(gate2.outp);

  property: SPEC (AG AF gate1.outp) & (AG AF ~gate1.outp);
}

module inverter(inp) {
  INPUT inp: boolean;
  outp: boolean;

  init(outp):= 0;
  next(outp):= ~inp;

  FAIRNESS running;
}
```

output of `gate1` must be infinitely often zero and must be infinitely often one. In fact, without a fairness constraint, this specification is false, since the system is not forced to execute a process infinitely often e.g. `gate2` would continuously execute. Hence, the output of a given gate may remain constant, regardless of changes of its input.

Figure 4.6 visualizes a snapshot of the corresponding transition system. As clearly shown by the figure, the Kripke model $\mathcal{M}$ is represented by a *disjunction* of the component (i.e. instance) Kripke models $\mathcal{M}_i$ [87], due to the interleaving semantics:

$$\mathcal{M} = \bigvee_i \mathcal{M}_i$$

Figure 4.6 sufficiently clarifies that the transition relation of $\mathcal{M}$ is made of the modules' transition relation as follows [87]:

$$\delta = \bigvee_i \delta_i \text{ where } \delta_i = (v_i' \iff f_i) \wedge (\bigwedge_{j \neq i} (v_j' \iff v_j))$$

Some state variables v_i' are given new values as determined by f_i, while the remaining variables just keep their old value.

So this time, the elements of the Kripke model are as follows: $\sigma = \{S0, S1, S2, \ldots\}$; $\sigma_0 = \{S0 = \{S0_{gate1}, S0_{gate2}, S0_{gate3}\}\}$; $\lambda(S0) = \{gate1.inp$

$= 0, gate1.outp = 0, gate2.inp = 0, gate2.outp = 0, gate3.inp = 0, gate3.outp = 0\}$; ....

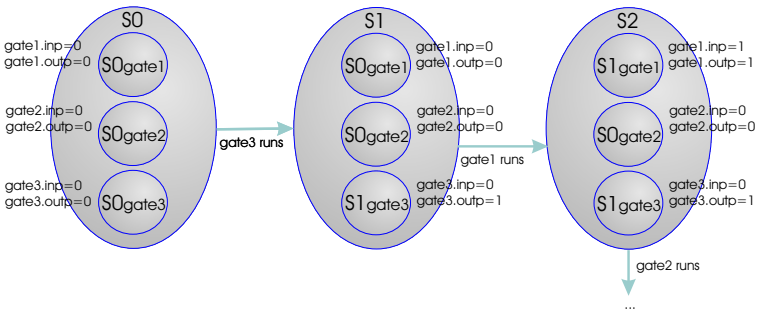


Figure 4.6: Partial Kripke Model of the Introductory Example

4.6 Methodology Extended

The architecture (or methodology) of our verification tool does not change dramatically. Instead of working with a standalone statechart, the tool peforms the necessary transformations on a set of communicating statecharts, which still have been formatted using the XMI exchange syntax (still has to be implemented). Now, it is the set of communicating statecharts that has to be mapped to a CaSMV model, which still represents a Kripke model that satisfies the semantics formally defined above. Chapter 9 applies the verification methodology to an industrial benchmark.

4.7 From a System to a Kripke Model $\mathcal{M}$

Communication between objects causes several statecharts to be present in the model of the embedded system. Some of these objects are active objects, while others are passive ones. This causes objects (statecharts) to be grouped together in separate threads of control, which operate interleaved to each other.

For the specification of larger systems, CaSMV models can be divided into several modules; which can be regarded as subroutines of the model. Each module can be reused several times. As illustrated in Section 4.5, modules can be parameterized, and each module can have its own local variable declarations, its own initialization and its own transition relation; thus its own Kripke structure. The modules are instantiated in a `main` module

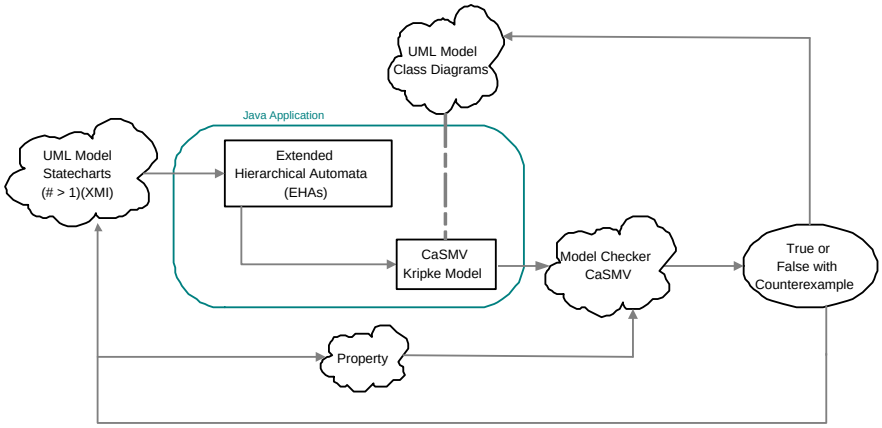


Figure 4.7: Methodology & Tool Architecture

similar to state variable declarations. The `main` module is what CaSMV uses to construct the Kripke model of the whole system. It has a special meaning in the same way that it does in the C programming language.

Since several objects are now grouped together in several threads, the problem is how to correctly define a Kripke model $\mathcal{M}$ for such complex systems. We say that (as in Section 4.5)

$$\begin{aligned} \mathcal{M} &= \bigvee_i \mathcal{M}_i \\ &= \bigvee_i (\sigma_i, \sigma_{0_i}, \delta_i, \lambda_i) \end{aligned}$$

with $\mathcal{M}_i$ the Kripke model of a thread. This is due to the fact that a thread shares the characteristics of a `module`. A visualization of the real lookings of $\mathcal{M}$ is provided by Figure 4.8.

The set of states σ of the model will follow from the way threads are encoded into CaSMV. Once having a suitable representation, CaSMV is capable to construct σ as the union of σ_i (This is done in the `main` module as mentioned previously.). The latter one refers to the set of states that belongs to a separate thread of the system. In Section 4.8, it is explained how σ_i really looks like. Of course, something similar can be said of σ_0, δ, and λ. You may again notice that states are not explicitly labeled, but they are implicitly labeled by their corresponding state-variable valuations.

Section 4.7.1 clarifies the encoding of the several threads the system is made of. Of course, a word on inter-threaded communication is given here as well,

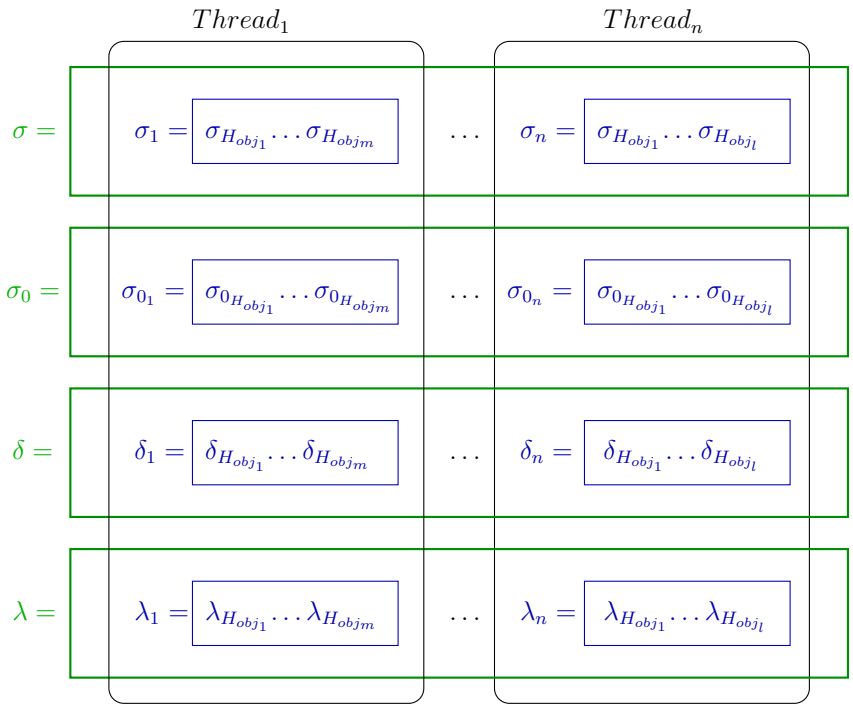


Figure 4.8: Kripke Model of a System

since the communication plays an important role to return a correct encoding scheme, which defines σ. Sections 4.7.2-4.7.3 discuss the set of initial states and the transition relation respectively.

4.7.1 The Set of States σ

The fundamental concept taken as our starting point is that of the thread of control. The embedded system is specified in terms of threads which are associated to the reception of several kinds of events. The behavior of each thread (as a collection of its containing objects' behavior) is reflected into a different `module` (see Section 4.8), which in turn is instantiated as a `process` in the `main` module (see Rule 4.1). This will instruct CaSMV to execute the threads interleaved.

Rule 4.1 (Thread Declaration). *Let the embedded system be divided into a collection of threads, called Threads. Then,* $\forall$ ***thr*** $\in$ *Threads, a variable is*

declared in the main module as follows:

***thr**: process thr_active(params);*

params: a list of formal parameters;
thr_active: a module named after the active object.

As an example, let us apply Rule 4.1 to the Rail Car System. The corresponding Kripke model consists of two threads (see Example 4.2) which is reflected into two separate modules, as illustrated in Code Listing 4.2. The list of parameters will be defined later on.

Code Listing 4.2 Thread of Control Declarations

```
module main() {
  ...;
  thr1: process car(params1);
  thr2: process term(params2);
  ...;
}

module car(params1) {
  ...;
  FAIRNESS running;
}

module term(params2) {
  ...;
  FAIRNESS running;
}
```

How to invoke inter-threaded communication? We just use a very straightforward strategy. The sender of an event is given the responsibility to accumulate the event into the corresponding queue of the receiver's threads. A module only has access to such queues when they are declared as global variables; thus when they are defined inside the `main` module (see Code Listing 4.3). Obviously, the declarations of both queues are performed using Rule 3.3.

It is worthwhile to note that the name of each event is preceded by the name of the reacting object. Only then, the dispatcher is able to serve every event to the correct target object.

As mentioned enough already, statechart transitions are treated in a run-to-completion manner. Thus, a transition will *freeze* in mid-execution,

Code Listing 4.3 Event Declarations

```
module main() {
  ...;
  thr1_event_queue: array 0..(SIZE-1) of {
                      s_crs_engage, ..., NotDefined
                    };
  thr1_call_queue : array 0..(SIZE-1) of {
                      c_car_departAck, ..., NotDefined
                    };

  thr1_event_tail    : boolean; thr1_call_tail    : boolean;
  thr1_event_overflow: boolean; thr1_call_overflow: boolean;

  thr2_event_queue: array 0..(SIZE-1) of {
                      s_hnd_departReq, ..., NotDefined
                    };
  thr2_call_queue : array 0..(SIZE-1) of {..., NotDefined};

  thr2_event_tail    : boolean; thr2_call_tail    : boolean;
  thr2_event_overflow: boolean; thr2_call_overflow: boolean;

  thr1: process car(params1);
  thr2: process term(params2);
  ...;
}
```

waiting for actions to complete. Moreover, it is required that all parts of a transition must be fully executed before the statechart becomes stable and the system can respond to another event. In this context, the difference between ordinary events (signals, interactions, flows) and synchronous calls is extremely important. Harel and Gery [50] wrote the following:

> *...when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations leading back to the same object instance is illegal, and an attempt to execute it will abort.*

Put in another way, after sending out a synchronous message, the calling

thread is blocked and deblocked when a reply is returned[2]. To integrate the (de)blocking mechanism into CaSMV, additional global variables, `block` variables and `origin_queues`, are injected into the Kripke model (see Rule 4.2).

Rule 4.2 (Blocking Declaration). *Let the embedded system be divided into a collection of threads, called Threads. Then,* $\forall$***thr*** $\in$ *Threads, if at least one object of thr possibly sends out a synchronous message, a block variable is declared in the main module as follows:*

thr_*block: boolean;*

Additionally, $\forall thr \in Threads$*, if at least one object of thr possibly responds to a synchronous message, an origin_queue is defined in the main module as follows:*

thr_*origin_queue: array 0..(SIZE-1) of {Obj1, . . . , Objn, NotDefined};*

with $\{Obj1, \ldots, Objn\}$ *the set of objects with whom a synchronous communication is performed.*

The value of a `block` variable evaluates to true at the moment a call event is generated by a specific transition. Otherwise the thread of control has not yet sent out a synchronous message or a reply has already been returned. Obviously the caller is given the responsibility to set the truth value whereas the callee must falsify it again, of course, at the appropriate moments.

Each synchronous message originates from another object. Therefore, to reset a `block` variable properly, additional `origin_queues` accumulate in the Kripke model. Figure 4.9 explains the relationship between the `call_queue` and the `origin_queue` of the same threads of control.

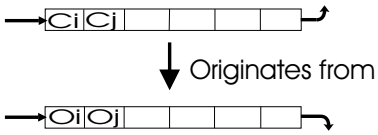


Figure 4.9: Relationship between Call Queue and Origin Queue

To state it simply, `block` variables denote active and sleeping threads. Their values make it possible to restrict the `running` variables in such a way that

[2]It is impossible that objects residing in the same thread of control synchronously communicate with each other.

the calling thread is frozen and therefore taken out of consideration in the interleaved model of computation (see Rule 4.3). When all processes get stuck (starvation due to cyclic calls), their stutter rules will be activated (see Section 4.8). Then a counterexample is constructed.

Rule 4.3 (Sleeping Declaration). *Let the embedded system be divided into a collection of threads, called Threads. Then,* $\forall \mathbf{thr} \in Threads$*, if at least one object of thr possibly sends out a synchronous message, the running variable of the corresponding module thr_active(params) is influenced as follows:*

$$INVAR\ \mathbf{thr}.running \rightarrow (\sim \mathbf{thr}\_block \mid activate);$$

with

$$activate:\ boolean;$$
$$activate := \bigwedge_{thr \in Threads} \mathbf{thr}\_block;$$

An illustration of blocking and sleeping declarations is given in Code Listing 4.4 applied on the Rail Car System. The statechart `CarHandler` responds to a synchronous message `CarHandler_departReq` somewhere along its execution path. Since the active object `Car` issues this request, it is placed in the origin queue of the receiving thread `thr2`.

Code Listing 4.4 Blocking and Sleeping Declarations

```
module main() {
  ...;
  thr1_block       : boolean;
  thr1_origin_queue: array 0..(SIZE-1) of {..., NotDefined};

  thr2_block       : boolean;
  thr2_origin_queue: array 0..(SIZE-1) of {
                       carObj, ..., NotDefined
                     };

  activate: boolean;
  activate:= thr1_block & thr2_block;
  ...;

  INVAR thr1.running ---> (~thr1_block | activate);
  INVAR thr2.running ---> (~thr2_block | activate);
}
```

4.7.2 Initial States σ_0

As before, initial states contain the initial values for system variables. The computation tree (Section 1.6.3) is obtained by unwinding the Kripke model from the initial states.

Threads are specified as processes in the model, thus they do not need to be initialized since each process defines a Kripke model, as part of the whole model, in turn.

Each queue is initialized using a strategy derived from Rule 3.5. To represent *open models* event queues are possibly initialized as non-empty queues. This is because interaction events are still allowed to accumulate in these queues. On the other hand, call stacks and origin stacks are always empty before execution starts since call events definitely are generated by the system. As everything happens automatically in the Rail Car System, it can be considered as a closed model, leading to queue initializations as shown in Code Listing 4.5.

Code Listing 4.5 Queue Initializations

```
init(thr1_event_queue)   := [NotDefined: i = 0..(SIZE-1)];
init(thr1_event_tail)    := 0;
init(thr1_event_overflow):= 0;

init(thr1_call_queue)   := [NotDefined: i = 0..(SIZE-1)];
init(thr1_origin_queue) := [NotDefined: i = 0..(SIZE-1)];
init(thr1_call_tail)    := 0;
init(thr1_call_overflow):= 0;

init(thr2_event_queue)   := [NotDefined: i = 0..(SIZE-1)];
init(thr2_event_tail)    := 0;
init(thr2_event_overflow):= 0;

init(thr2_call_queue)   := [NotDefined: i = 0..(SIZE-1)];
init(thr2_origin_queue) := [NotDefined: i = 0..(SIZE-1)];
init(thr2_call_tail)    := 0;
init(thr2_call_overflow):= 0;
```

At the very beginning of a run, no process will respond to a synchronous message. Thus the initial value of each block variable is false.

4.7.3 Transition Behavior δ

An embedded system can be specified in terms of threads. Each thread consist of a collection of statechart diagrams. Put in another way, thread behavior is based on the behavior of its objects. Thus each module, as representation of threads, will have a transition relation (see Section 4.8) that conforms both the thread behavior and the RTC-step semantics. The transition relation of $\mathcal{M}$ is built as the union of the modules' transition relation, the same way as it was built in Section 4.5.

4.8 From a Thread to a Kripke Model $\mathcal{M}_\mathbf{i}$

It is already known that $\mathcal{M}$ is built upon the union of $\mathcal{M}_i$, which are the Kripke models belonging to the threads the system is made of. The question now is how a thread will be mapped to a Kripke model $\mathcal{M}_i$.

Each thread bundles a set of objects: at most one active object and possibly at least one passive object (see Section 4.3). Each object has its own statechart and therefore also its own extended hierarchical automata, which we now denote as H_{obj}. In Chapter 3, we have mapped a single hierarchical automaton H_{obj} to a model $\mathcal{M}_{H_{obj}}$. Knowing this, we say that

$$\begin{aligned} \mathcal{M}_i &= (\sigma_i, \sigma_{0_i}, \delta_i, \lambda_i) \\ &= \bigvee_k (\sigma_{H_{obj_k}}, \sigma_{0_{H_{obj_k}}}, \delta_{H_{obj_k}}, \lambda_{H_{obj_k}}) \end{aligned}$$

What does the formula precisely mean? As you can see on Figure 4.8, the set of states σ_i is a combination of the set of states $\sigma_{H_{obj_k}}$ of each automaton that belongs to the thread. Or, a state of σ_i is a union of configurations i.e. one configuration of each automaton H_{obj_k}. This is because for a single EHA, we have mapped its configurations to σ (Section 3.4).

The set of initial states σ_{0_i} is built the same way. It refers to the initial configurations of the thread, of course, which is defined by the initial configurations of its containing objects.

The transition relation δ_i again covers the STEP relation, which either is the STEP relation defined in Definition 4.2 or the one defined in Definition 4.3. It completely depends on the total amount of objects gathered in the thread.

As before, the construction of the Kripke models $\mathcal{M}_i$ is split up into three different blocks. Section 4.8.1 explains how σ_i is encoded in CaSMV. A word on the initial states σ_{0_i} is given in Section 4.8.2. Finally, Section 4.8.3 specifies the translation of the STEP relation into CaSMV constructs.

4.8.1 The Set of States $\sigma_\mathbf{i}$

A module is a bundle of definitions. Much like a subroutine, a module may have formal parameters. When creating an instance of the module, actual variables are plugged in for the formal parameters, thus linking the module instance to the program. Most often the formal parameters of a module are declared to be either inputs or outputs. Inputs are expected to be assigned outside the module, whereas outputs are expected to be assigned inside the module. Sometimes a formal parameter can be declared als both an input and an output. Whether or not formal parameters are assigned inside or outside a module, they need to be declared in the module anyway.

Each module contains a list of parameters. This list is defined based on the communication among objects of threads and the fact that senders of events have the responsibility to place the events in the queues of the receiving threads (see Section 4.3.1).

Rule 4.4 applied to the Rail Car System gives a module heading as captured in Code Listing 4.6. We omit to give the declaration of these variables since it is completely the same as the declaration of these variables in the main module.

Rule 4.4 (Module's List of Parameters). *Let the embedded system be divided into a collection of threads, called Threads. Then, the list of parameters for a particular thread* $\mathbf{thri} \in Threads$ *contains the following elements:*

$$\left.\begin{array}{l} \mathbf{thrj}\_event\_queue \\ \mathbf{thrj}\_event\_tail \\ \mathbf{thrj}\_event\_overflow \end{array}\right\} \textit{if } \exists o_i \in \mathbf{thri}, o_j \in \mathbf{thrj} : async(o_i, o_j)$$

$$\left.\begin{array}{l} \mathbf{thrj}\_call\_queue \\ \mathbf{thrj}\_call\_tail \\ \mathbf{thrj}\_call\_overflow \\ \mathbf{thrj}\_origin\_queue \\ \mathbf{thrj}\_block \end{array}\right\} \textit{if } \exists o_i \in thri, o_j \in \mathbf{thrj} : sync(o_j, o_i)$$

$async(o_i, o_j)$: object o_i communicates asynchronously with object o_j;
$sync(o_j, o_i)$: object o_j communicates synchronously with object o_i;

$$\left.\begin{array}{ll} \mathbf{thri}\_event\_queue & \mathbf{thri}\_call\_queue \\ \mathbf{thri}\_event\_tail & \mathbf{thri}\_call\_tail \\ \mathbf{thri}\_event\_overflow & \mathbf{thri}\_call\_overflow \\ & \mathbf{thri}\_origin\_queue \\ & \mathbf{thri}\_block \end{array}\right\} \ldots \textit{of its own}$$

Code Listing 4.6 Module Headings

```
module Car(thr1_event_queue, thr1_event_tail, thr1_event_overflow,
           thr2_event_queue, thr2_event_tail, thr2_event_overflow,
           thr1_call_queue, thr1_call_tail, thr1_call_overflow,
           thr2_call_queue, thr2_call_tail, thr2_call_overflow,
           thr1_origin_queue, thr1_block,
           thr2_origin_queue, thr2_block,
          ) {

  /** Declaration of the parameters **/
  thr1_event_queue: array 0..(SIZE-1) of {
                      s_crs_engage, ..., NotDefined
                    };
  ...;
};
```

Obviously, the configuration of a thread is built upon the configurations of the containing standalone statecharts. Automata and attributes are declared using Rule 3.1 and Rule 3.2 respectively, but the object name is integrated in the declaration. As an example, we give some declarations for the `Car module` (see Code Listing 4.7).

Code Listing 4.7 Automata and Attribute Declaration of Module Car

```
car_st_Root       : {..., operating, ..., StateMachineError};
car_st_Operating  : {..., departure, ..., NotActive};
car_st_Departure  : {commHandler, commCruiser, end, NotActive};
car_st_commHandler: {init, waitExit, syncExit, NotActive};
car_st_commCruiser: {..., syncCruiser, ..., NotActive};

crs_st_Root: {..., engaged, disengaged, StateMachineError};
```

4.8.2 Initial States σ_{0_i}

The initial configuration of a thread is built upon the initial configuration of its containing standalone statecharts. Thus, it is sufficient and straightforward to initialize each statechart separately by using the rules defined in Section 3.6. Note that it is not needed to initialize the parameters of the module, since these parameters are initialized in the `main` module of $\mathcal{K}$.

4.8.3 Transition Behavior $\delta_\mathbf{i}$

Definition 4.3 guides the way the transition relation of each template will be constructed[3]. Every statechart running on a particular thread of control satisfies the operational semantics of a standalone statechart. As a consequence, the transition behavior of each module will have the raw structure as illustrated in Code Listing 4.8.

Code Listing 4.8 Raw Structure of a Module's Transition Behavior

```
/* Transition Behavior

   for each object (one active, several passive) in the thread:
     1. first phase of the progress rule
            (object_progress_auto)
   for each object (one active, several passive) in the thread:
     2. second phase of the progress rule
              (object_progress_call)
   for each object (one active, several passive) in the thread:
     3. third phase of the progress rule
              (object_progress_trigger)
   for all objects in the thread define a global stutter rule

**/
```

Not surprisingly, Definition 4.3 forces us to use additional variables and a `case` statement to achieve the accurracy of the semantics (Rule 4.5). Let us explain this a little bit more in detail. The intent of the variable `last` is to indicate the last object (statechart) that has performed some activity during a RTC-step. Clearly, we have to define a transition relation for this variable. It is initialized with the object that executes right from the start. As we will see later on, its value is updated in each branch of the transition behavior.

Rule 4.5 (Additional Variables). *Let Obj be the set of objects running on the same thread of control* ***thr****, and* $Object_k$ *(*$\in Obj$*) the object that starts the execution of the thread. To achieve a correct operational semantics inside the module* ***thr****, two additional variables are inserted into* ***thr*** *as follows:*

[3]Definition 4.2 can be seen as a special case of Definition 4.3.

$last$: $\{Obj\} \cup \{NotDefined\}$;
$object\_progress$: $\{Obj\} \cup \{NotDefined\}$;

$init(last) := Object_k$;

$object\_progress := case$ {
  ...
  $last = Object_i$ & $Object_i\_progress\_auto$: $Object_i$;
  ...
  $\sim (\bigvee_{Object_i \in Obj} Object_i\_progress\_auto)$: $NotDefined$;
  $default$: $\{Object_i\_progress\_auto\ ?\ Object_i\}$;
};

It is the purpose of the variable `object_progress` to indicate the object's statechart that has to be evaluated until a stable state configuration is reached. Its value is calculated based upon the value of the variable `last`. If the last active object still has an instable configuration (indicated by Object$_i$ _progress_auto), the value of `object_progress` is set to the name of this last object. This is because the RTC-step of each object must be finished before making progress in the statechart of another object. However, if the last object has reached a stable configuration, then we have to check if there are other objects that became instable due to the progress made in the last active statechart. If this is the case, the value of `object_progress` is set non-deterministically. It is only when all the objects have reached a stable configuration that a new event will be served to one of the objects.

Both variables are very useful to build the `case` statement of the step relation (see Rule 4.6). As a simple extension of the standalone step relation (Rule 3.7.1), the first branches of the statement correspond with the first phase of the progress rule ($\xrightarrow{prog}_{\varepsilon}$). The next branches correspond with the second and third phase of the progress rule ($\xrightarrow{prog}_{q_c[0]}$, $\xrightarrow{prog}_{q_e[0]}$) respectively. Rule 4.6 also shows a slight change in the stuttering rules. Now, the statecharts are forced to reside in the `StateMachineError` state when either a fault (`error`) or a deadlock (`activate`) has occurred.

Rule 4.6 (Thread's Step Relation Representation). *Let Obj be the set of objects running on the same thread of control* `thr`*. Then, $\forall Object_i \in Obj$, the step relation of a module looks as follows:*

```
case {
   ...
   object_progress = Object_i & ~error : {
      ...;
      next(last):= Object_i;
   };
   ...;

   ...;
   Object_i_progress_call & ~error:{
      ...;
      next(last):= Object_i;
   };
   ...

   ...;
   Object_i_progress_trigger & ~error:{
      ...;
      next(last):= Object_i;
   };
   ...
   error | activate : {...};
   default : {...};
};
```

Each `branch` statement performs the necessary updates (local states, attributes, and queues) based on the transitions that execute during a time step. The *next* statements that represent these changes are specified in exactly the same way as it is done in Section 3.7. It is even possible to use these rules to achieve communication between the objects. To be complete, Code Listing 4.9 illustrates some of these updatings for both types of communication.

Remember that a transition freezes in mid-transition at the moment a synchronous communication is invoked. It is impossible to represent such a freezing in CaSMV, but it can be simulated using *dummy* states. Basically, what this means is that right after sending out a call event, the statechart resides in this *dummy* state, and not in the target state of the transition. Thus, transitions generating call events have to be rewritten, following the rewrite rule as illustrated in Figure 4.10.

Code Listing 4.9 Partial Thread Update

```
module Car(...) {
  /* Declarations */
  car_t1:= in_Init;
  car_t2:= in_synchExit & in_synchCruiser
                        & (thr1_call_queue[0]=c_car_departAck);

  /* Transition Behavior (Partial) */
    car_progress_auto & ~error: {
      for (i = 0; i < SIZE; i = i + 1) {
        case {
          car_t1 : next(thr2_event_queue[i]):= s_hnd_departReq;
          default: next(thr2_event_queue[i]):= thr2_event_queue[i];
        };
      };
      for (i = 0; i < SIZE; i = i + 1) {
        case {
          car_t2 : next(thr1_event_queue[i]):= s_crs_engage;
          default: next(thr1_event_queue[i]):= thr1_event_queue[i];
        };
      };
    };

    car_progress_call & ~error: {
      if ( ((i + 1)<event_tail) & (i + 1)<SIZE) {
        next(thr1_call_queue[i]):= thr1_call_queue[i+1];
        next(thr1_origin_queue[i]):= thr1_origin_queue[i+1];
      };
      else { ...; };

      next(thr2_block):= case {
        car_t2 : 0;
        default: thr2_block;
      };
    };
  };
```

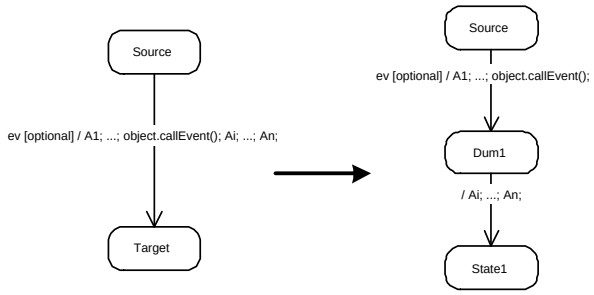


Figure 4.10: Rewrite Rule for Synchronous Communications

4.9 Template Embedded Models Extended

We can now extend the template embedded model of a standalone statechart (Section 3.8) to cover communicating statecharts. The usefulness of the template remains the same.

4.10 Conclusion

This chapter (and the previous one) has presented a method for the verification of UML statechart models generated in the development process of embedded systems. The method allows the automated verification of behavioral requirements through model transformation and application of the model checker CaSMV. This approach must be used to avoid logical design errors in a (relatively) early design phase, before the implementation and execution begins. Chapter 9 applies the verification methodology to an industrial benchmark.

CHAPTER 5

Specification Correctness

We all have our time machines.
Some take us back, they're called memories.
Some take us forward, they're called dreams.
Jeremy Irons.

Verification and validation is critical and costly for high-assurance systems. In finite-state verification, model checkers examine finite-state machines, representing embedded software systems, to look for errors (such as deadlocks) in the design. Errors are defined as violations of requirements; as violations of properties of the system.

Even though many formal specification techniques are available to verify various properties for embedded systems, it takes much effort to specify properties using *temporal logic*, because embedded systems have a complex nature, and a merely intuitive understanding is not sufficient to draw conclusions about their behavior.

In reactive systems, temporal logics are used to describe and reason about sequences of states evolving in time. This is important because such systems continually respond to their environment, and are mostly designed not to terminate. It is not the purpose of this chapter to show how to write temporal logic formulas. Instead, it is assumed that the professional engineer is capable to define such formulas in relation to the statecharts. However, before giving them to the CaSMV embedded model, they need to be transformed, because the embedded model sometimes contains *dummy* states not present in the statecharts. This chapter will explain the reason and the solution for these transformations. But first, some mathematical background of temporal logics, as used by the model checking technique, is given.

5.1 Model Checking of Temporal Logic

Temporal logic model checking [26] is one of the most popular and well studied paradigms for the formal verification of hardware systems, concurrent systems, reactive systems, etc. In temporal logic model checking, the system is modeled as a Kripke model $\mathcal{M} = (\sigma, \sigma_0, \delta, \lambda)$ (Definition 3.1). The correctness property that needs to be verified on the given Kripke structure is specified in terms of a temporal logic formula φ. In general, a model checking problem is a problem of checking whether a given model satisfies a given property; it is a problem of determining the validity of the formula on specific states of the transition system (denoted as $\mathcal{M} \models \varphi$).

Temporal logic is a famous formal specification language for use with a model checker. It is a nonclassical logic that enable us to formulate and verify propositions about situations dynamically changing in time. As an example, consider the statement: "The door of the dishwasher is open." Though its meaning is constant in time, the truth value of the statement can vary in time. Sometimes the statement is true, and sometimes the statement is false, but the statement is never true and false simultaneously. In a temporal logic, statements can have a truth value which can vary in time. Contrast this with an atemporal logic, which can only handle statements whose truth value is constant in time.

Technically, a temporal logic is a logic augmented with temporal modalities to allow specification of event orders in time, without having to introduce time explicitly. For example, a temporal logic with the modalities *always* and *eventually* will be able to specify the following property: "for all future moments in which p holds there will be a future moment in which q holds". Whereas traditional logics can specify properties relating to the initial and final states of terminating systems, a temporal logic is better suited to describe on-going behavior of non-terminating and interacting (reactive) systems.

5.2 The Language of Temporal Logic

Temporal logic, used by model checking tools, serves to formally state properties concerned with the executions of a system.

As must be known by now, a path (or an execution) refers to a sequence of states (Definition 5.1). Temporal logic uses *atomic propositions* to make statements about the states. Atomic propositions are elementary statements which have a well-defined truth value in a given state. For example, statements like "**open**", "**in_phase_1**" are atomic propositions.

These propositions are assembled into a set denoted $Prop = \{P_1, P_2, \ldots\}$. Obviously, a proposition P_i is defined as true in a state q if and only if $P_i \in \lambda(q)$ [12].

Additionally, temporal logic uses *boolean combinators* like the negation ($\neg$), the conjunction ($\wedge$), the disjunction ($\vee$), the logical implication ($\Rightarrow$), the double implication ($\Leftrightarrow$) and the constants **true** and **false**. This allows us to define more complex statements relating various simpler sub-formulas [12]. For example, $\textbf{error} \Rightarrow \neg\textbf{warm}$.

Lastly, *temporal combinators* allow one to speak about the sequencing of the states along an execution, rather than about states individually. Some examples: P_i states a property of the current state, whereas $\mathbf{X}P_i$ states that the *next state* (temporal combinator **X** for "next") satisfies P_i. The statement $P_i \vee \mathbf{X}P_i$ states that P_i is satisfied in either the current state, or in the next state, or in both states. Of course, temporal combinators can be arbitrarily nested [12].

The two most popular logic frameworks for system verification are:

- *linear time*: at each moment there is only one possible future
- *branching time*: time has a tree like nature in which, at each instant, time may split into alternative courses representing different futures

As Figure 5.1 illustrates, the difference lies in the semantics of the time structure. In a logic of linear time, temporal modalities are provided for describing events along a single time line. Whereas in a logic of branching time, the modalities reflect the branching nature of time by allowing quantification over possible futures.

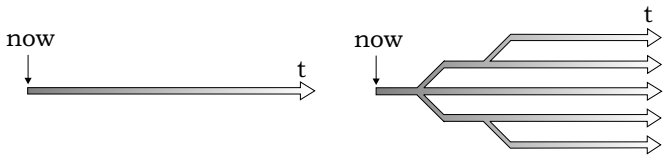


Figure 5.1: Linear Time versus Branching Time

In the following definitions of these logics, we refer to *paths* and *reachable states*.

Definition 5.1 (Path). *A path in $\mathcal{M}$ is a finite or infinite sequence of states $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0, s_i \rightarrow s_{i+1}$.*

We denote individual states of π by their subindices: for instance, π_0 is the first state of the path, and π_i is the $i+1th$ state of the path. A superindexed path π^i denotes a suffix of π that starts at π_i. In particular, $\pi^0 = \pi$.

Definition 5.2 (Reachable State). *A state s is called reachable in $\mathcal{M}$, if there exists a finite path π in $\mathcal{M}$ such that $\pi_0 \in \sigma_0$, and the last state of π is s. In other words, there exists a path from some initial state to the state s $\mathcal{M}$.*

5.2.1 Branching Time Logic

In branching time temporal logic [26], each moment in time may have several possible futures. Branching time temporal logics are interpreted over structures that can be viewed as trees, each describing the behavior of the possible computation of a non-deterministic system. In such logics, the temporal operators quantify over the paths that are possible from a given state.

One such branching logic is the Computation Tree Logic (CTL), which extends propositional logic with unary temporal operators (**EX** and **AX**, **EF** and **AF**, **EG** and **AG**) and binary temporal operators (**EU** and **AU**)[1]. The informal semantics of these operators is given in Section 1.6.3.

Syntax The CTL formulas are syntactically described as follows:

$$
\begin{array}{lll}
\varphi, \psi ::= & P_1 \mid P_2 \mid ... & \text{(atomic propositions)} \\
& \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \ldots & \text{(boolean combinators)} \\
& \mid \mathbf{EX}\varphi \mid \mathbf{EF}\varphi \mid \mathbf{EG}\varphi \mid \mathbf{E}[\varphi \mathbf{U} \psi] & \text{(temporal combinators)} \\
& \mid \mathbf{AX}\varphi \mid \mathbf{AF}\varphi \mid \mathbf{AG}\varphi \mid \mathbf{A}[\varphi \mathbf{U} \psi] & \text{(temporal combinators cont.)}
\end{array}
$$

Of course, this is an abstract grammar. In practice, each tool dealing with temporal formulas will allow parentheses, and will have its own operator priority conventions. As well, each tool will have its specific set of atomic propositions and combinators.

Semantics Let us turn to the semantics of CTL, i.e. given a model $\mathcal{M}$ and a CTL formula φ, how to determine whether $\mathcal{M}$ satisfies φ ($\mathcal{M} \models \varphi$)? To do that, we have to define a satisfaction relation for states and then for models. Naturally, the interpretation of CTL formulas is defined directly on the set of states, but the notion of a path (Definition 5.1) is also used. In general,

[1] Normally, a temporal logic also provides a release operator R, but this operator is omitted here, because CaSMV does not support it.

for the statements given below $\mathcal{M}, s \models \varphi$ means that a state s in the model $\mathcal{M}$ satisfies the formula φ.

$$
\begin{array}{lcl}
\mathcal{M}, s \models p & \Leftrightarrow & p \in \lambda(s) \\
\mathcal{M}, s \models \neg\varphi & \Leftrightarrow & M, s \not\models \varphi \\
\mathcal{M}, s \models \varphi \wedge \psi & \Leftrightarrow & M, s \models \varphi \text{ and } M, s \models \psi \\
\mathcal{M}, s \models \varphi \vee \psi & \Leftrightarrow & M, s \models \varphi \text{ or } M, s \models \psi \\
 & & \\
\mathcal{M}, s \models \mathbf{EX}\varphi & \Leftrightarrow & \exists s' : s \rightarrow s' \text{ and } M, s' \models \varphi \\
\mathcal{M}, s \models \mathbf{AX}\varphi & \Leftrightarrow & \forall s' : s \rightarrow s' \text{ implies } M, s' \models \varphi \\
 & & \\
\mathcal{M}, s \models \mathbf{EF}\varphi & \Leftrightarrow & \exists \pi : \exists i \geq 0 : \pi_0 = s \text{ and } M, \pi_i \models \varphi \\
\mathcal{M}, s \models \mathbf{AF}\varphi & \Leftrightarrow & \forall \pi : \exists i \geq 0 : \pi_0 = s \text{ implies } M, \pi_i \models \varphi \\
 & & \\
\mathcal{M}, s \models \mathbf{EG}\varphi & \Leftrightarrow & \exists \pi : \forall i \geq 0 : \pi_0 = s \text{ and } M, \pi_i \models \varphi \\
\mathcal{M}, s \models \mathbf{AG}\varphi & \Leftrightarrow & \forall \pi : \forall i \geq 0 : \pi_0 = s \text{ implies } M, \pi_i \models \varphi \\
 & & \\
\mathcal{M}, s \models \mathbf{E}[\varphi\ \mathbf{U}\ \psi] & \Leftrightarrow & \exists \pi : \pi_0 = s \text{ and} \\
 & & \exists i \geq 0 : \mathcal{M}, \pi_i \models \psi \text{ and } \forall j < i : \mathcal{M}, \pi_j \models \varphi \\
\mathcal{M}, s \models \mathbf{A}[\varphi\ \mathbf{U}\ \psi] & \Leftrightarrow & \forall \pi : \pi_0 = s \text{ implies} \\
 & & \exists i \geq 0 : \mathcal{M}, \pi_i \models \psi \text{ and } \forall j < i : \mathcal{M}, \pi_j \models \varphi
\end{array}
$$

CTL Expansion Rules When the transition relation is total, CTL operators can be expanded as follows:

$$
\begin{array}{lcl}
\mathbf{EF}\varphi & = & \varphi \vee \mathbf{EX}\ \mathbf{EF}\varphi \\
\mathbf{AF}\varphi & = & \varphi \vee \mathbf{AX}\ \mathbf{AF}\varphi \\
 & & \\
\mathbf{EG}\varphi & = & \varphi \wedge \mathbf{EX}\ \mathbf{EG}\varphi \\
\mathbf{AG}\varphi & = & \varphi \wedge \mathbf{AX}\ \mathbf{AG}\varphi \\
 & & \\
\mathbf{E}[\varphi\ \mathbf{U}\ \psi] & = & \psi \vee (\varphi \wedge \mathbf{EX}\ \mathbf{E}(\varphi\ \mathbf{U}\ \psi)) \\
\mathbf{A}[\varphi\ \mathbf{U}\ \psi] & = & \psi \vee (\varphi \wedge \mathbf{AX}\ \mathbf{A}(\varphi\ \mathbf{U}\ \psi))
\end{array}
$$

For instance, note that in order for $\mathbf{EF}\varphi$ to hold at state s, either φ may hold at s, or there must be an edge from state s to state t such that $\mathbf{EF}\varphi$ holds at t. Hence we can write $\mathbf{EF}\varphi = \varphi \vee \mathbf{EX}\ \mathbf{EF}\varphi$.

5.2.2 Linear Time Logic

In linear time temporal logic (LTL), each moment in time has a unique possible future. The logic only deals with the set of executions and not with

the way these are organized into a tree. Linear temporal logic formulas are therefore interpreted over linear sequences and are regarded as specifying the behavior of a single computation of a system; a linear time formula cannot examine alternative executions which split off from this one at each time step where a non-deterministic choice is possible.

Linear time temporal logic extends propositional logic with unary temporal operators (**X**, **G**, and **F**) and a binary temporal operator (**U**). The informal semantics of these operators is given in Section 1.6.3.

Syntax The LTL formulas follow the formal grammar described as follows:

$$\begin{array}{lll} \varphi, \psi ::= & P_1 \mid P_2 \mid ... & \text{(atomic propositions)} \\ & \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \ldots & \text{(boolean combinators)} \\ & \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi \mathbf{U} \psi & \text{(temporal combinators)} \end{array}$$

Semantics The validity of formulas in LTL is originally defined on a path rather than on an individual state, and we say that an LTL formula φ holds in a state s iff φ holds on all paths π starting from s. This is denoted as $M, \pi \models \varphi$.

$$\begin{array}{lcl} \mathcal{M}, \pi \models p & \Leftrightarrow & p \in \lambda(\pi_0) \\ \mathcal{M}, \pi \models \neg\varphi & \Leftrightarrow & M, \pi \not\models \varphi \\ \mathcal{M}, \pi \models \varphi \wedge \psi & \Leftrightarrow & M, \pi \models \varphi \text{ and } M, \pi \models \psi \\ \mathcal{M}, \pi \models \varphi \vee \psi & \Leftrightarrow & M, \pi \models \varphi \text{ or } M, \pi \models \psi \\ \\ \mathcal{M}, \pi \models \mathbf{X}\varphi & \Leftrightarrow & M, \pi^1 \models \varphi \\ \\ \mathcal{M}, \pi \models \mathbf{G}\varphi & \Leftrightarrow & \forall i \geq 0 : \mathcal{M}, \pi^i \models \varphi \\ \\ \mathcal{M}, \pi \models \mathbf{F}\varphi & \Leftrightarrow & \exists i \geq 0 : \mathcal{M}, \pi^i \models \varphi \\ \\ \mathcal{M}, \pi \models \varphi \ \mathbf{U} \ \psi & \Leftrightarrow & \exists i \geq 0 : \mathcal{M}, \pi^i \models \psi \text{ and } \forall j < i : \mathcal{M}, \pi^j \models \varphi \end{array}$$

5.3 The Problem

As mentioned in Chapters 3- 4, the behavior of embedded systems is specified within a set of statechart diagrams. From this, a semantical CaSMV model is constructed, which serves as the Kripke model $\mathcal{M}$ during the verification process.

Temporal logic is used to express requirements, with respect to the statechart diagrams, that are not globally valid (in all states), but temporally

or from a certain point onwards. CaSMV verifies these requirements through symbolic model checking. When this process gets stuck, CaSMV constructs a counter model, which is a path through the model not satisfying the desired properties. The counter model in an infinite sequence of states, i.e. one or more states are accessed infinitely often.

Unfortunately, the verification process must be guided very carefully, since there are some syntactical differences between the statecharts and the Kripke structures. The proposed Kripke model of statechart diagrams denotes an ordering on the states that allows us to link requirements to the corresponding state(s).

Transitions in UML statecharts are atomic and cannot be interrupted. Sometimes, it is possible that a transition is broken up into additional transitions and states through the translation to a CaSMV Kripke model (e.g. because of synchronous calls, Section 4.8.3). As a direct consequence, requirements are verified in these additional (dummy) states too. It cannot be a suprise that this can result in a wrong verification process. For example, in Figure 5.2 a next state of state $S1$ is state $S5$ and not the additional state D.

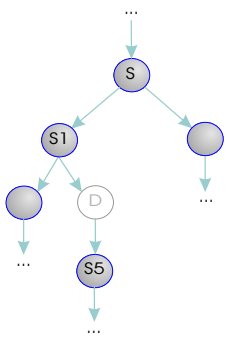


Figure 5.2: Execution Trace with Additional States

To avoid confusion in the formulation and the verification of requirements, we need to transform the temporal formulas so that they cope with possible additional states. This way, additional states are made transparent during the verification process. For example, if the software engineer wants to know something about a next state, we must assume that he means a next state in his behavioral model (= statecharts), because he is not aware of the dummy states that are introduced in the CaSMV Kripke models.

5.4 The Solution

The most obvious solution is to use a *clock operator* [40] to make additional states transparent for the verification process. The clock operator allows applying a formula only to states satisfying some condition. Unfortunately, only the *ForSpec* [6] language from Intel and the *Accelera PSL* language (a.k.a. Sugar 2.0 [11]) both have such a construct. Since CaSMV does not support clock operators, the formulas have to be translated to achieve the same results as with the clock operators. Basically, the interpretation of formulas have to be changed. In doing so, we make the verification process perserve the behavioral semantics of objects.

5.4.1 CTL Transformations

We present a relatively simple solution using the CTL semantics. Some words about the notations: $\mathcal{I}(\varphi)_{\mathcal{M}}$ refers to the interpretation of φ made in the Kripke model $\mathcal{M}$ (= translated statechart diagrams); *dummy* holds all dummy states present in the model so $dummy \equiv \bigvee_i dummy_i$.

EXφ Transformation The property φ must hold in a next state that is not a dummy state. Defining a new intepretation for such a formula is a little bit tricky:

$$\mathcal{I}(\mathbf{EX}\varphi)_{\mathcal{M}} ::= \mathbf{EX}(\mathbf{E}(dummy\ \mathbf{U}\ (\neg dummy \wedge \varphi))$$

With such an interpretation, φ would only be true, if from the current state zero or more intermediate states are visited, until a non-dummy state is reached that satisfies φ. The interpretation is best understood using the CTL expansion rule of the **EU** operator (Section 5.2.1). Using the latter definition, we first verify whether the next state is a non-intermediate one that satisfies φ. If not, the next state can be an intermediate one for which the process has to be repeated. If the next state is not an additional state, then the next state of another execution path is examined.

AXφ Transformation Analogously, we have the following interpretation:

$$\mathcal{I}(\mathbf{AX}\varphi)_{\mathcal{M}} ::= \mathbf{AX}(\mathbf{E}(dummy\ \mathbf{U}\ (\neg dummy \wedge \varphi))$$

EFφ Transformation Following the CTL semantics, **EF**φ states that it is possible (by following a suitable execution) to have φ some day. This interpretation allows that φ holds in a dummy state of some execution path,

which is of course not allowed. To guarantee that φ is not verified in such additional states, we have to change the interpretation as follows:

$$\mathcal{I}(\mathbf{EF}\varphi)_{\mathcal{M}} ::= \mathbf{EF}(\varphi \wedge \neg dummy)$$

AFφ Transformation Analogously as before, the interpretation becomes:

$$\mathcal{I}(\mathbf{AF}\varphi)_{\mathcal{M}} ::= \mathbf{AF}(\varphi \wedge \neg dummy)$$

EGφ Transformation **EG**φ means that there exists an execution path along which φ always holds, i.e. in every state of the path. The UML semantics forbids to interrupt a transition, and thus the semantics forbids to verify φ in the intermediate states because φ can be false in some of these states. A new interpretation

$$\mathcal{I}(\mathbf{EG}\varphi)_{\mathcal{M}} ::= \mathbf{EG}(\varphi \vee dummy)$$

makes sure that these dummy states are correctly skipped. With such an interpretation available, the model checker now verifies whether φ holds in all non-dummy states along an execution path, as requested.

AGφ Transformation Analogously as before, the interpretation becomes:

$$\mathcal{I}(\mathbf{AG}\varphi)_{\mathcal{M}} ::= \mathbf{AG}(\varphi \vee dummy)$$

E(φ U ψ) Transformation Obviously, φ and ψ must be satisfied in non-intermediate states. Thus,

$$\mathcal{I}(\mathbf{E}(\varphi\ \mathbf{U}\ \psi))_{\mathcal{M}} ::= \mathbf{E}((\varphi \vee dummy)\ \mathbf{U}\ (\psi \vee dummy))$$

A(φ U ψ) Transformation Analogously,

$$\mathcal{I}(\mathbf{A}(\varphi\ \mathbf{U}\ \psi))_{\mathcal{M}} ::= \mathbf{A}((\varphi \vee dummy)\ \mathbf{U}\ (\psi \vee dummy))$$

5.4.2 LTL Transformations

Deriving new interpretations for LTL formulas is done similarly as it is done for CTL formulas. Therefore, we just give the interpretations without further explanations.

$$\begin{array}{lcl}
\mathcal{I}(\mathbf{X}\varphi)_{\mathcal{M}} & ::= & \mathbf{X}(dummy\ \mathbf{U}\ (\neg dummy \wedge \varphi)) \\
\mathcal{I}(\mathbf{F}\varphi)_{\mathcal{M}} & ::= & \mathbf{F}(\varphi \wedge \neg dummy) \\
\mathcal{I}(\mathbf{G}\varphi)_{\mathcal{M}} & ::= & \mathbf{G}(\varphi \vee dummy) \\
\mathcal{I}(\varphi\ \mathbf{U}\ \psi)_{\mathcal{M}} & ::= & (\varphi \vee dummy)\ \mathbf{U}\ (\psi \vee dummy)
\end{array}$$

5.5 Specifications in CaSMV

An *assertion* is a condition that must hold true in every possible execution of the program. Assertions in CaSMV are written in linear time temporal logic, that makes it possible to succinctly state proportions about the relation of events in time. A declaration of the form

```
assert p;
```

where `p` is a linear temporal formula, means that every execution of the program must satisfy the formula `p`. An execution that does not satisfy the formula is called a failure of the program. On the other hand, a declaration of the form

```
SPEC p;
```

denotes that `p` is a branching temporal formula.

To write down the new formula interpretations in CaSMV, it is useful to introduce *abstract* auxiliary signals:

```
abstract <signal> :  <type>;
```

Such signals can be part of the specification, can be part of the proof, but do not belong to the system being verified. An abstract signal is useful to cover the *dummy predicate*, as used in the previous sections (Rule 5.1). It can be seen as a shortcut.

Rule 5.1 (Abstract Dummy Signal). *The 'dummy predicate', as used in the transformed temporal formulas, is inserted into CaSMV as follows:*

$$\begin{array}{l} \textit{abstract dummy: boolean;} \\ \textit{dummy} := \bigvee_i \ \textit{in\_dummy\_i;} \end{array}$$

in_dummy_i: true whenever the statechart resides in one of the additional states.

5.6 Methodology Extended

Software developers are allowed to describe important properties, in English, about (embedded) software systems with the intention of specifying what the eventual system will be expected to provide. Quite often the requirements of a system follow simple patterns.

A property specification pattern describes the essential structure (=scope) of some aspect of a system's behavior. As an example, the scope "global"

means that the requirement should always hold. Using such property patterns, expectation phrases are transformed into temporal logic formulas through the assistant that guides the user in writing properties. The property writing assistant is based on the pattern scheme proposed by Dwyer et al. [39].

As shown by Figure 5.3, our tool integrates (at least, this is the intention) a versatile assistant that guides the software developer in writing properties to be verified using temporal logic. Moreover, the tool automatically transforms the temporal formulas such that additional states are correctly dealt with.

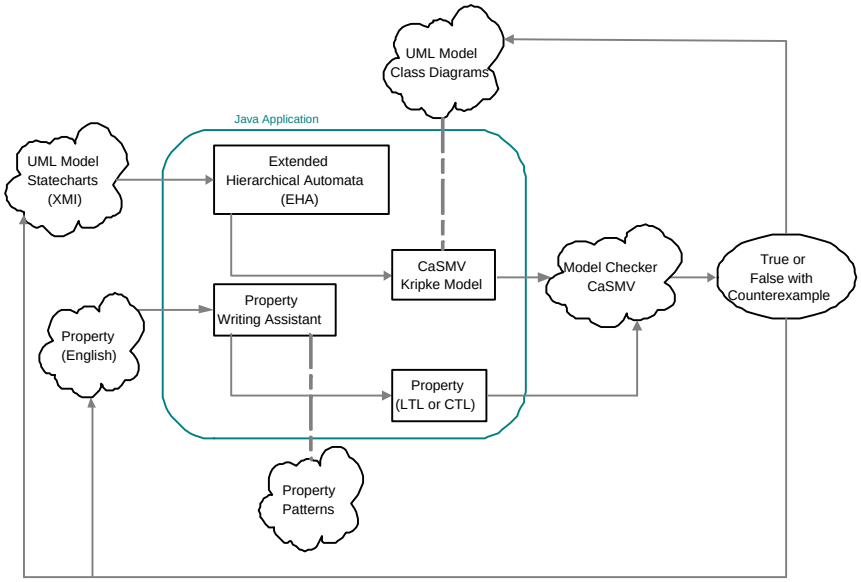


Figure 5.3: Methodology & Tool Architecture

It must be clear that the verification is carried out in such a way that the developers do not need knowledge of either formal languages or temporal logic to be able to take advantage of its potential; something which has traditionally been difficult to overcome when deciding on the use of formal methods; the engineer only needs knowledge of UML and the system studied, the tool automatically obtains a CaSMV formal representation from a textual XMI representation.

CHAPTER 6

Protocol Conformance

The piece, when it is over,
is not what is made, but how it is made.
Andrew Kuntz.

If you are familiar with object-oriented methods, you will be aware of the concept of a class and an interface. A *class* represents an entity of a given system and provides a piece of system functionality, whereas an *interface* is a variation of a class in the sense that it only provides a definition of system functionality. Interface classes are used by classes that claim to implement them.

The UML allows that classes use *behavioral state machines* — also called statecharts — to describe their piece of system functionality. The new version of the standard, UML 2.0, allows that interfaces use *protocol state machines* to focus only on allowable sequences of behavior invocations on a class, but, without having to show its behavior.

A problem now arises: given a protocol state machine and a behavioral statechart, that should be an implementation of it, how to verify that the implementation meets the specification? This problem is called *protocol conformance* verification.

This chapter addresses this problem in model-driven software design. The contribution is to provide a methodology, which is based on refinement mappings, to automatically verify protocol conformance. It will be shown, that the verification procedure, described in Part 2, can easily be extended with this new methodology. This helps to turn UML into a powerful and interesting tool for the development of critical systems.

6.1 Two Kinds of State Machines

Behavior in UML 2.0 is defined as a specification of how its context classifier changes state over time [96]. *Behavior* is an abstract metaclass and the specification of behavior can take a number of forms, as described in its subclasses. A variety of behavior specification mechanisms are provided by UML, such as *Statemachine*, *Activity*, *Usecase* and *Interaction*.

State machines define a set of concepts that can be used for modeling behavior through finite state transition systems. In general, a state machine is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change [129].

In addition to expressing the behavior of a part of a system, state machines can also be used to express usage protocols of a part of a system. Based on this, UML 2.0 differentiates between two kinds of state machines: *behavioral state machines* and *protocol state machines*.

6.1.1 Behavioral State Machines

In UML 2.0, behavioral state machines ($\mathcal{BSM}$), extend traditional finite automata by adding hierarchy (= state refinement to contain another automaton), concurrency and communication. They are hierarchical automata associated to UML objects (a class instance) to model their behavior. Each $\mathcal{BSM}$ gives an abstract view of all the desired behaviors of an object in its lifecycle. In fact, a behavioral state machine is completely equivalent with the elder UML statecharts, used throughout this manuscript so far. It can be seen as a view that is concerned with what an object *must* do.

6.1.2 Protocol State Machines

A protocol state machine is always defined in the context of a classifier (mostly an interface). In its simplest form, a $\mathcal{PSM}$ (see Definition 6.1) is a state diagram in standard UML notation whose transitions are labeled by events (call event, signal event, time event, completion event) and do not have actions; i.e. refusing any behavioral implementation.

Definition 6.1. *A UML protocol state machine is a 6-tuple* $\mathcal{PSM} = \langle \sigma, \delta, E, PREC, POSTC, s^0 \rangle$ *where* σ *is a finite set of states,* E *is a finite set of events,* $PREC$ *is a finite set of preconditions,* $POSTC$ *is a finite set of postconditions,* $\delta \subseteq \sigma \times PREC \times E \times POSTC \times \sigma$ *is a finite set of transitions, and where* s^0 *is the initial state. (The definition can be extended with mappings to the sources, targets, and labels of the transitions.)*

The big difference in notation between a behavioral machine and a protocol machine is in the transition line between the states. Now, the syntax for transitions is:

[pre-condition] event / [post-condition]

These transitions have no actions. Instead, they only have effect descriptions. This means that a $\mathcal{PSM}$ is side-effect free. Pre (post) conditions have the same syntax as guard conditions.

Obviously, a $\mathcal{PSM}$ captures the *triggering view* of an objects behavior. It presents the possible and the permitted transitions on the instances of its context classifier, by specifying the consummation order of the events and the states through which an object progresses during its life. A $\mathcal{PSM}$ acts as an independent specification and defines *what the instances of its context classifier can do*. The triggering view is also a requirement to the environment external to the state machine: it is legal to send this event to an instance of the context classifier only under the conditions specified by the protocol state machine.

Note that each protocol state machine can also be rewritten into an extended hierarchical automaton, in the same way as can be done for statecharts. Therefore, we omit to give any further details.

6.2 Motivating Example

The motivating example used throughout this chapter, is based on a bounded `Stack`. A `Stack` is a data structure that works on the principle of *Last In First Out* (LIFO). Data items are *pushed on* and *popped from* the top of the stack. Since bounded stacks have a maximum size, it is an error to push items onto a full stack. We also get an error if we try to pop items from an empty stack.

An abstract data type (ADT) specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not specify the implementation of the operations. Each ADT denote the operations you need without having to think at the same time about how the methods are performed. The stack as ADT is defined by specifying the operations (*push, pop, isfull, isempty*) that can be performed on it. The specification is called an *interface*.

UML 2.0 gives us the opportunity to present the interface by a protocol state machine showing the way how the stack is used in practice. A possible protocol state machine for the stack ADT is given in Figure 6.1. Four states are used to define the lifecycle of a stack: `Empty, Loaded, Full,`

and `Exception`. Events are used to model the pushing and popping of data items.

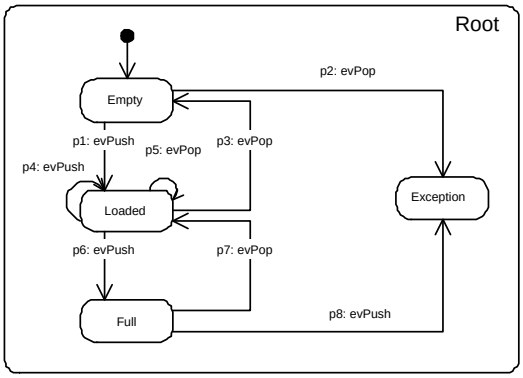


Figure 6.1: Protocol State Machine of a Stack

There are many uses for stacks: providing support for recursive procedure calls, searching structures, computation, and so on. Depending on the use, the methods of the stack can be implemented in many different ways, each leading to different stack classes (array based, list based, ...). A possible overall behavior of such a stack class is given by Figure 6.2, which shows that the behavior of the stack is defined in a reactive manner. Obviously, the behavioral transitions contain some *guards* – a predicate expression associated with an event – which might change depending on how the stack is implemented[1]. Note that the specified behavior considers the stack when it is almost empty or almost full.

Obviously, a protocol machine specifies what a behavioral machine is allowed to do at any given moment. At this point, the designer only has to worry about whether the implementation – the behavioral state machine – is correct in accord with the interface – the protocol state machine. This problem is known as protocol conformance verification. It is the purpose of this chapter to show how this problem can be tackled during the early phases of system design.

[1]Since we are working in the design phase, the transitions do not yet contain corresponding method calls.

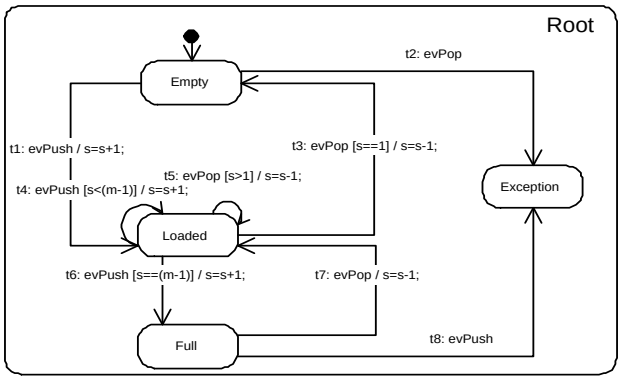


Figure 6.2: Behavioral State Machine of a Stack

6.3 Protocol Conformance

The UML Superstructure 2.0 Specification [96] explains that there are relationships between the classifier being the context of the specific $\mathcal{BSM}$ and the classifier being the context of the $\mathcal{PSM}$. Generally the former specializes or realizes the latter. In UML 2.0, protocol conformance means that every rule and constraint specified for the general $\mathcal{PSM}$ applies to the specific $\mathcal{BSM}$. This is augmented by specifying that the $\mathcal{PSM}$ can be redefined into the $\mathcal{BSM}$. Clearly, the behavioral view of an object is not independent of its triggering view. Intuitively, the ordered collection of stimuli received by an object's statechart must exist as a sequence of events in its corresponding protocol state machine. That is, every behavior of an object's statechart is also a behavior of its protocol statechart. If not, then obviously, the interface class is wrongly implemented. In order to verify the consistency between both state machines, we need a more formal definition of protocol conformance between a $\mathcal{PSM}$ and a $\mathcal{BSM}$ and adapt a method presented in [59]. This is where refinement mappings, as considered by Abadi and Lamport [82], will show their use.

6.3.1 Refinement Mappings (or Simulations)

The existence of a refinement mapping proves that a machine implements a given specification. You may know that refinement mappings are the functional cousin of Milner's *simulation relations*. Milner introduced them for the purpose of comparing programs [95]. The simulation guarantees that every behavior of a structure is also a behavior of its abstraction. However

the abstraction might have behaviors that are not possible in the original structure. Technically, a simulation (or a refinement) is a function (or a mapping) R between the states of a low-level specification $\mathcal{S}_1$ and a high-level specification $\mathcal{S}_2$ that satisfies conditions [26] as

$$(s, v) \in R \wedge s \rightarrow_{\mathcal{S}_1} s' \Rightarrow \exists v' : v \rightarrow_{\mathcal{S}_2} v' \wedge (s', v') \in R$$

(If a low-level state s and a high-level state v are related, and $\mathcal{S}_1$ can make a transition from s to s', then there exists a matching transition in $\mathcal{S}_2$ from v to a state v' that is related to s'.) The existence of such a mapping implies that any behavior that can be exhibited by $\mathcal{S}_1$ can also be exhibited by $\mathcal{S}_2$. To summarize things, a refinement mapping consists of a state mapping, a steps mapping and implicitly a behavioral mapping as well.

It can be of no suprise that a refinement mapping gives us everything we need to automatically decide on protocol conformance. Each $\mathcal{PSM}$ is equivalent to an abstract specification $\mathcal{PSM}_a$ while each $\mathcal{BSM}$ defines a concrete specification $\mathcal{BSM}_c$ of some classifier. Verifying on protocol conformance is now equal to proving that a low-level specification (i.e. $\mathcal{BSM}_c$) correctly implements a high-level specification (i.e. $\mathcal{PSM}_a$), using a refinement mapping between a $\mathcal{BSM}$ and a $\mathcal{PSM}$.

A refinement mapping, denoted as R_f, from a $\mathcal{BSM}$ to a $\mathcal{PSM}$ is a function $f : \sigma_{\mathcal{BSM}} \rightarrow \sigma_{\mathcal{PSM}}$ that provides the following mappings:

1. state mapping: $\forall s \in \sigma_{\mathcal{BSM}} : f(s) \in \sigma_{\mathcal{PSM}}$
2. steps mapping: $\forall (s_i, s_j)_{e[g]/a} \in \delta_{\mathcal{BSM}} : (f(s_i), f(s_j))_{[pre]e[post]} \in \delta_{\mathcal{PSM}}$
3. behav. mapping: $xs \in Beh(\mathcal{BSM}) \Rightarrow f^w(xs) \in Beh(\mathcal{PSM})$

State Mapping

There can be no formal connection (and therefore no refinement mapping) between both state machines unless the designer has specified a correspondence relation between the states of both statecharts. To simplify things, we assume that a $\mathcal{PSM}$ presents the possible and permitted states through which an object progresses during its life i.e. a $\mathcal{BSM}$ cannot reside in states not present in its corresponding usage protocol. Summarized, the state mapping is given by a state transformer (Definition 6.2) that assumes a 1-to-1 correspondence between the states of $\mathcal{BSM}$ and the states of $\mathcal{PSM}$. Obviously, the initial state s^0_{BSM} corresponds to the initial state s^0_{PSM}.

Definition 6.2. *The set of states that an object may have during its life is fully defined in its $\mathcal{PSM}$:*

$$\forall s \in \sigma_{\mathcal{BSM}} : \exists! s \in \sigma_{\mathcal{PSM}} : f(s) = s'$$

Steps Mapping

Analogously, Definition 6.3 specifies the mapping of the steps. Each step is equal to a transition.

Definition 6.3. *A behavioral transition, with label **event[guard]/actions**, from state s_i to state s_j is legal iff the corresponding $\mathcal{PSM}$ defines a protocol transition from state s_i to state s_j with the label **[pre]event/[post]**.*

$$\forall (s_i, s_j)_{e[g]/a} \in \delta_{\mathcal{BSM}} : (f(s_i), f(s_j))_{[pre]e[post]} \in \delta_{\mathcal{PSM}}$$

This means that a behavior transition may exist iff there exists a protocol transition with the same source, target and triggering event.

Still, it is not required that every $\mathcal{PSM}$'s state/transition has a counterpart state/transition in its redefined $\mathcal{BSM}$ because a $\mathcal{PSM}$ specifies all the capabilities of a classifier, not all of which may be used in a particular system. That is, $\sigma_{\mathcal{BSM}} \subseteq \sigma_{\mathcal{PSM}}$ and $\delta_{\mathcal{BSM}} \subseteq \delta_{\mathcal{PSM}}$.

Behavioral Mapping

As already mentioned, protocol conformance means that every behavior of an object's statechart is also a behavior of its protocol statechart. Thus, Definition 6.4 implicitly defines the requested mapping of behaviors (= sets of all sequences of transitions) i.e. $xs \in Beh(\mathcal{BSM}) \Rightarrow f^w(xs) \in Beh(\mathcal{PSM})$.

Definition 6.4. *Let P be a $\mathcal{PSM}$ and B defined for a class c. B conforms to P with respect to the initial state s^0 if and only if whenever*

$$s^0 \rightarrow^* s' \overset{event[guard]/actions}{\rightarrow} s''$$

(that is, a transition is triggered from some state s' in B that is reachable from the initial state of B) we have a corresponding counterpart transition in P

$$s^0 \rightarrow^* s' \overset{[pre]event/[post]}{\rightarrow} s''$$

where both the pre and the post condition evaluate to true.

The refinement mapping R_f can now be used to prove whether the low-level specification correctly simulates the high-level one. If so, then the $\mathcal{BSM}_c$ implements the $\mathcal{PSM}_a$ or the $\mathcal{BSM}_c$ conforms the $\mathcal{PSM}_a$. Obviously, there must exist some verification technique (and corresponding tool) that uses R_f to prove the protocol conformance during system design. Additionally, it is useful if the technique finds some interesting counter examples, helping the modeler to develop the design of her/his system.

6.4 Methodology

The refinement methodology enforces the verification process of the protocol conformance to proceed through phases:

Phase 1 The first phase automatically decides whether the set of states of the $\mathcal{BSM}_c$ is indeed a subset of the one of the corresponding $\mathcal{PSM}_a$. If not, the low-level specification contains states that are not present in the high-level one. At this point, the developer is informed about her/his mistake(s). Of course, if a fault occurs, it is impossible to have protocol conformance between both machines and continuing with the other phases does not make sense.

Example 6.1. *The set of states used to define the behavioral stack machine (see Figure 6.2) is fully defined in its corresponding usage protocol (see Figure 6.1). More precisely, the sets are in this case equal.*

Phase 2 Analogously to the first phase, the second phase verifies whether each behavioral transition has a corresponding counter protocol transition. If not, then the mapping of the transitions is not correctly followed, and the designer is again informed about his mistake(s).

Example 6.2. *It is easy to see that each behavioral transition in Figure 6.2 has a corresponding counter protocol transition in Figure 6.1.*

Phase 3 The last phase proves the satisfaction of the behavioral mapping. This phase shows that every implementation behavior is allowed by its definition, specified in the $\mathcal{PSM}_a$. Of course, this shall be done by an exhaustive search respecting the run-to-completion step semantics of statecharts. At this stage, in order to efficiently perform this proof, a sophisticated tool is needed. We propose the Cadence SMV model checker.

6.4.1 Behavioral Mapping Auxiliary

The Cadence SMV system [86] – used to model check behavioral properties (see Chapters 3-4) – provides an approach that is geared to proving that an abstract model is implemented by some more detailed system model. The notion of correctness is defined in terms of refinement maps that relate signaling behavior at suitable points in the implementation with events occurring in the abstract model. The verification is based on a circular compositional rule that allows us to assume that one map (as a temporal property) holds true while verifying another map, and vice versa. The construct *layer* is used to provide the refinement maps. A layer is defined as a collection of abstract signal definitions. These are expressed as assignments in the same way the implementation is defined. Thus, inside a layer, transition relations are specified. More precisely, higher layers offer an abstracted description of the system. They build its specification, and describe *what* it *should* do. The lower layers implement the higher layers, and describe *how* the system actually does what it should do.

Example 6.3. *Let's consider a very simple example [86] of a specification and an implementation of a finite state machine, as written down in Code Listing 6.1. The specification states that* `x` *is* `0` *at time* `t = 0`*, and states that the value of* `x` *at time* `t + 1` *is* `1` *if* `x` *is* `0`*, and else is either* `0` *or* `1`*. Clearly, the implementation satisfies the specification which is verified by an exhaustive search of the state space of the implementation. Note that the transition relation inside the layer defines the protocol state machine in Figure 6.3, while the transition relation outside the layer corresponds to the behavioral state machine in Figure 6.4.*

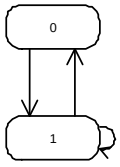


Figure 6.3: Example $\mathcal{PSM}$

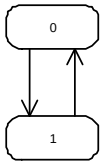


Figure 6.4: Example $\mathcal{BSM}$

Using CaSMV gives us the opportunity to tie the verification of behavioral properties together with the protocol conformance proof. Both proofs happen through the model checking technique. Note, that CaSMV only has to verify the behavioral mapping since the state and transition transformer connect both state machines purely syntactically. Obviously, syntax checking can be

Code Listing 6.1 Sample CaSMV Protocol Verification

```
module main () {
   x: boolean;

   /* The specification */

   layer spec: {
     init(x) := 0;
     if (x=0) next(x) := 1;
     else next(x) := {0,1};
  }

   /* The implementation */
       init(x) := 0;
       next(x) := ~x;
}
```

performed before enabling the refinement verification; it can be done without using CaSMV.

Intuitively, to prove the protocol conformance, CaSMV executes the $\mathcal{BSM}_c$ following the execution semantics as close as possible and checks it against the information covered in the refinement mapping defined inside a layer. If no fault is found, we are allowed to say that the $\mathcal{BSM}_c$ complies with its corresponding $\mathcal{PSM}_a$. Otherwise, the model checker returns a useful counterexample, helping the user to locate and to eliminate the design problem.

Following Definition 6.4, the layer must contain the transition relation (see Example 6.3) of the $\mathcal{PSM}_a$. The transition relation outside the layer (see Example 6.3) is the one of the $\mathcal{BSM}_c$ resulting in the structure shown in Code Listing 6.2.

Doing so, the behavioral mapping is correctly defined and the model checker is capable of proving the conformance all by itself, as wanted. The main obstacle to face here is the construction of the transition relations. We will now show how to solve this problem using the `stack` as an example (see Section 6.2).

Behavioral Transition Relation outside a Layer

Chapters 3-4 defined a template structure in the CaSMV language [86] in order to be able to model check some behavioral properties of a system

Code Listing 6.2 Sample CaSMV Structure of Conformance Verification

```
module main() {

   -- high-level specification, triggering view
   layer protocol : {
     -- PSM's transition relation
   }

   -- low-level specification, behavioral view
   -- BSM's transition relation

};
```

under development. To do so, each statechart, contained in the model, is transformed into an extended hierarchical automaton, without loss of generality. Additionally, the statechart's execution semantics is rewritten into phases making up the total transition relation of behavioral state machines. Instantiating the template structure on the behavioral stack machine (Figure 6.2) results in the CaSMV representation, as shown in Code Listing 6.3, which of course is simplified to the parts of main interest. The relation between the code and the graphical representation of the behavior is straightforward.

Protocol Transition Relation inside a Layer

The transition relation for a $\mathcal{PSM}$ must be specified in such a way that Definition 6.4 is correctly used within the verification process. Informally, Definition 6.4 says that the execution of a behavioral transition leads to the execution of some counter protocol transition.

Code Listing 6.4 shows the CaSMV representation of the protocol state machine (Figure 6.1). You can easily understand that `init/next` statements are used to specify the refinement mapping. The `init` statement mentions the state, the behavioral statechart has to reside in at time `t = 0`. The `next` statements represents the allowable state changes at time `t + 1`. Each protocol transition has a unique identifier, again used to define the condition under which a transition is able to take the state change.

As like the behavioral transition relation, the protocol transition relation is split up in several phases or blocks as well. This is motivated by the fact that the execution of a behavioral transition is dictated by entering the several phases. Moreover, the refinement mapping is more transparent

Code Listing 6.3 Specification of Stack's Behavioral View

```
...;
t4 := in_Loaded & event_queue[0] = evPush & (size < (m-1));
t6 := in_Loaded & event_queue[0] = evPush & (size == (m-1));
...;
/* Initialization of Behavioral View */
init(st_root) := Empty;

/* Total Transition Relation of Behavioral View */
case {
  ...;
  progress_auto & ~error: {  };
  progress_trigger & ~error: {
     next(st_root) :=
       case {
         t1: Loaded;
         t2: Exception;
         t4: Loaded;
         t6: Full;
         t3: Empty;
         t5: Loaded;
         t7: Loaded;
         t8: Exception;
         default : st_root;
       };
      ...;
  };
  error: {
     next(st_root) := StateMachineError;
     ...;
  };
  default: {
     next(st_root) := st_root;
     ...;
  };
};
```

in the CaSMV representation. However, the `error` block is not integrated inside the layer since it seems unlogical for a $\mathcal{PSM}_a$ without behavioral implementations attached to it.

Example 6.4. *It is allowed to reach state* `Full` *whenever* `t7` *and* `p7` *are*

Code Listing 6.4 Specification of Stack's Triggering View

```
layer protocol: {
   ...;
   p4 := in_-Loaded & event_queue[0] = evPush;
   p6 := in_-Loaded & event_queue[0] = evPush;
   ...;
   /* Initialization of Triggering View */
   init(st_root) := Empty;

   /* Total Transition Relation of Triggering View */
   case {
     progress_auto & ~error: {  };
     progress_trigger & ~error: {
      next(st_root) :=
        case {
          p1 & t1: Loaded;
          p2 & t2: Exception;
          p4 & t4: Loaded;
          p6 & t6: Full;
          p3 & t3: Empty;
          p5 & t5: Loaded;
          p7 & t7: Loaded;
          p8 & t8: Exception;
          default : st_root;
        };
     };
   default: {
      next(st_root) := st_root;
   };
   };
};
```

enabled. Indeed, a behavioral transition, `t7`, is only allowed to execute whenever its abstract counter protocol transition, `p7` is executed as well. It is forbidden to reach state `Full`, if behavioral transition `t7` is enabled to execute, but the conditions of its abstract counter protocol transition are not fulfilled.

Inside the layer, every protocol transition is connected to its behavioral counterpart. Executing a low-level transition is only possible whenever a corresponding high-level transition is executed as well, as specified by Definition 6.4. This explains the reason for *conjuctions* inside the layer.

This is illustrated by Example 6.4. If we leave out the conjunction and only use the abstract transitions to define the transition relation inside the layer, strange things can happen, i.e. if behavioral transition `t7` is enabled to execute, the abstraction transition `p5` shall be executed as well instead of abstract transition `p7`. Indeed, transition `p5` has the same activation condition as the one of abstract transition `p7` and is evaluated/taken first in the case statement. Leaving out the behavioral transitions inside the layer leads to a bad implementation of Definition 6.4.

The proof

The stack example shows a layer as a collection of a single abstract signal definition *st_root*. Such an abstract definition entails a verification task — to show that every implementation behavior is allowed by this definition, to show that both machines are protocol conformant. Indeed, the model-checker verifies whether the low-level implementation of a signal is simultaneously consistent with its abstract definition for each possible behavior. The way this is achieved is the same as it is achieved in Example 6.3.

At time `t = 0` the model checker verifies whether the behavior machine is in the correct initial state. The stack machine clearly fulfills this requirement. Next, at time `t + 1`, each behavioral state change is verified against the state changes defined inside the layer. If at time `t + 1` the behavioral stack reaches a state that can never be reached by the layer, a counterexample is returned to the user. This is not the case for the stack example; the behavioral stack is protocol conformant with its protocol statechart.

6.5 Compositional Verification

Mostly, behavioral statecharts consist of several sequential submachines i.e. hierarchical sequential states or regions of concurrent states. The corresponding protocol statecharts contain the abstract version of these submachines, otherwise the machines are not protocol conformant. The presence of several submachines leads to several (abstract) signal definitions (inside) outside the layer respectively. Each abstract signal definition entails a verification task. This is known as decomposition. Now, the behavioral machine complies to the protocol machine, if each signal (= sequential submachine) is consistent with its abstract signal definition. Therefore, it must be possible to use decomposition in order to verify the refinement mapping.

Definition 6.5. *Let P be a $\mathcal{PSM}$ and let B be a $\mathcal{BSM}$ (with sub-automata) defined for a class c. B conforms to P if and only if every sub-automaton in B conforms to its abstract counterpart sub-automaton in P.*

Instead of using a single layer, decomposition allows us to use several layers to prove protocol conformance. Each layer will contain the abstract signal definition of a particular sub-automaton. Using several layers, instead of a single one, results in the compositional refinement verification. Each layer entails the verification task of the refinement mapping of sub-automata as Definition 6.5 requires.

6.6 Related Work

In the design of large systems, the designer often faces a trade-off as to which validation technique to use in order to guarantee that the final implementation will meet the specification. One such validation technique is *refinement verification*. This technique shows its usefulness in many fields. We mention a few of them.

As a first example, in [83] a new class of refinement maps has been introduced for pipelined machine verification. The result is that the use of such refinement maps leads to drastic improvements in verification times. As another example, refinement maps can also be used to refine abstract architectural models into more platform specific representations [9].

To come to the field of UML, the refinement concept for UML statecharts has been formalized in [91]. Here, refinement maps are defined in terms of configurations and simulations, which is slightly different from our refinement map definition. In [62] an extension of the Temporal Logic of Actions (TLA) is defined in order to identify adequate concepts of refinement for mobile UML state machines. Such an extension has the advantage to work with more complex refinement maps; our refinement maps are limited to the capabilities of CaSMV.

Important Usage Difference The verification of a system against a given property is done in two phases when using refinements. In a first phase, the coherence (=consistency) between the specification and the refinement is verified. If both are consistent with each other, the property is verified on the high-level model. Of course, the high-level model is usually much smaller than the low-level one. This means that a two-phase approach proves to be more efficient than a direct verification [88]. For our purposes, verifying a particular property does not benefit from this two-phase approach. This follows directly from the definition and the use of our refinement maps.

6.7 Conclusion

This chapter has shown that UML state diagrams may be used at two different levels of granularity during the behavioral design of applications; the behavioral view focuses on the reactivity of a class whereas the protocol view is known as the *object life cycle*. Additionally, we have provided a methodology for verifying the consistency between both views. This methodology is built around refinement mappings. The latter ones have

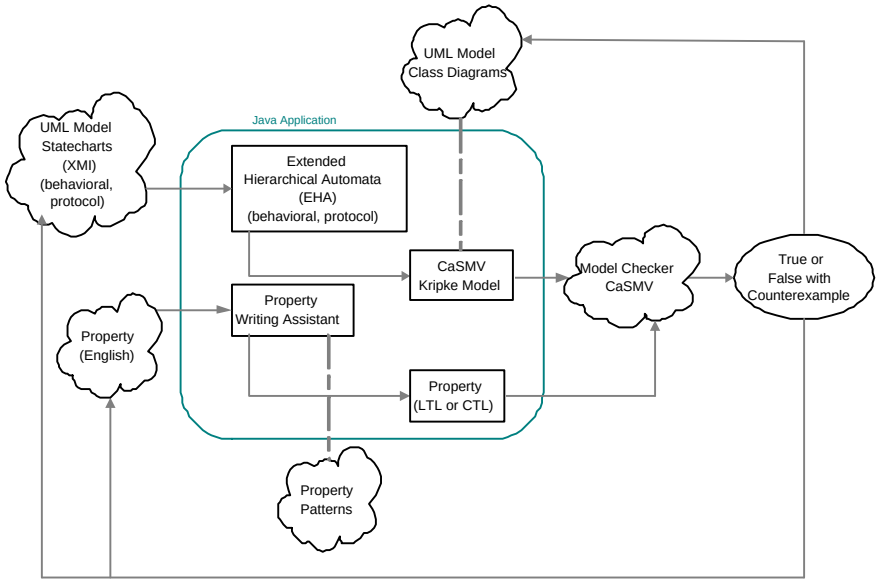


Figure 6.5: Methodology & Tool Architecture

proven themselves to be extremely useful to verify whether a class (as a $\mathcal{BSM}$) correctly implements its interface (as a $\mathcal{PSM}$), known as protocol conformance verification. Not suprisingly, the verification of behavioral properties can be done at the same time the consistency between both machines is verified. However, the exact translation from a protocol automaton to a layer of CaSMV is explicitly omitted, since it is almost similar to the construction of a behavioral model. For reasons to be complete, Figure 6.5 shows how our application deals with protocol verification (not yet implemented).

Part II

Optimization

CHAPTER 7

The Basics of Slicing Hierarchical Automata

Our greatest glory is not in never failing,
but in rising up every time we fail.
Ralph Waldo Emerson.

As formal verification becomes increasingly important in the industry as a part of the design process, there is a constant need for efficient verification techniques that are capable to deal with real-size applications. Unfortunately, our approach (as presented in Part 2) suffers from the so-called *State (Space) Explosion* problem that can arise when a system is composed of several subsystems. In this case, a finite state model with a number of states, which is exponential to the number of the component subsystems, can be generated. Systems that are highly dependent on data values share the same problem, producing a number of states exponential to the number of data variables.

Moreover, in order to modify a program, we are mostly not interested in its complete behavior but only in its behavior at a given point of interest. This means that we do not observe each complete program state but only a part of it. For instance, consider the case of debugging. When an error is observed, the programmer tries to extract that part of the program which is responsible for the erroneous behavior. This set of statements might be much smaller than the original one and therefore, it might be much easier to catch and to correct the error(s) when he only looks at this set.

The above observations motivate us to reduce the state space of UML statecharts using the method of program slicing [127]. This chapter will present the algorithm of [125] to reduce the state space in model checking by slicing extended hierarchical automata with respect to the temporal properties to be verified; it will serve as the basis for further research.

7.1 A Brief History of Program Slicing

Almost every programmer has endured the unpleasant task of sifting through hundreds of lines of code in order to find an error in just one. *Tedium* ad *nauseam*: there must be a better way. Enter program slicing.

Program slicing[1] is a pretty simple idea. Its fundamental concepts are easily grasped by novices and it applies to many problem domains; who doesn't want to help in restricting a complicated problem to the relevant focus of inquiry?

Program slicing is a decomposition technique which extracts program elements related to a particular computation. A *program slice* consists of those parts of a program that may (directly or indirectly) affect the values computed at some program point of interest, referred to as a *slicing criterion*.

Program slices were first proposed by Weiser [126, 127, 128]. He introduced the concept of *executable backward static* slices. *Executable* because the slice is required to be an executable program. *Backwards* because the slice consists of all program points that affect a given point in the program. Informally, a backward slice provides an answer to the question *which program statements potentially affect the computation of variable* `v` *at statement* `s`*?* And finally, *static* because they are computed as the solution to a static analysis problem i.e. without considering the program's input. Of course, each slicing algorithm must be *sound* as well, because the algorithm must not slice away parts of the program that affect the given slicing criterion.

Example 7.1. *The backward slice shown in Code Listing 7.1 indicates exactly which statements influence the output of variable* `i`.

Code Listing 7.1 A Backward Slice

```
void main() {
  1: int i = 1;
  2: int sum = 0;
  3: while (i<11) {
      4: sum = sum + i;
      5: i = i + 1;
  }
  6: printf("sum = %d", sum);
  7: printf("i = %d", i);
}
```

[1] For a complete history of program slicing, we refer to the survey in [112].

Weiser originally introduced slices by computing consecutive sets of indirectly relevant statements, according to *data flow* and *control flow* dependencies. He used a *control flow graph*, as an intermediate program representation for his slicing algorithm. An alternative approach to compute such backward slices was suggested by Ottenstein et al. [97]. They noted that backward slices could be efficiently computed using a *program dependence graph*, again as an intermediate program representation, by traversing the dependence edges backwards (from target to source).

Bergeretti & Carré were the first to define a notion for an *executable forward static* slice in [13]. Informally, a *forward* slice answers the question *which statements are affected by the value of variable* `v` *at statement* `s`*?*

Example 7.2. *The forward slice shown in Code Listing 7.2 indicates exactly which statements are influenced by the initialization of variable* `sum`*.*

Code Listing 7.2 A Forward Slice

```
void main() {
  1: int i = 1;
  2: int sum = 0;
  3: while (i<11) {
      4: sum = sum + i;
      5: i = i + 1;
  }
  6: printf("sum = %d", sum);
  7: printf("i = %d", i);
}
```

Finally, the exact terminology *dynamic program slicing* was first introduced by Korel and Larski [63]; a slice is computed for a particular fixed input. The availability of run-time information makes dynamic slices smaller than static slices, but limits there applicability to that particular input.

7.2 Some Background Material

This section gives some important definitions for program slices to be able to capture the computation of slices for hierarchical automata. As there are many forms of slices, the definitions are defined for a simple form; the static backward slice. Note that the other forms can be thought of as augmentations of this static form, and have forward counterparts.

As mentioned already, a slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point, referred to as a *slicing criterion* [112].

Definition 7.1 (Slicing Criterion). *A slicing criterion of a program* `Pr` *is a pair* $< s, V >$ *where* `s` *is a statement in* `Pr` *and* `V` *is subset of the variables in* `Pr`*.*

As the slice is simpler than the original program, e.g. the slice in Code Listing 7.1, yet contains all the statements which could possibly affect the final (and incorrect) value of the variable i, examining the slice will allow us to find the error faster than examining the original program.

Definition 7.2 (Program Slice). *The slice* `S` *of a program* `Pr` *with respect to a slicing criterion* $< s, V >$ *consists of only those statements of* `Pr` *needed to capture the behavior of* `V` *at* `s` *[16].*

Definition 7.3 (Executable Program Slice). *The slice* `S` *of a program* `Pr` *with respect to a slicing criterion* $< s, V >$ *is any executable program with the following properties [16]:*

- `S` *can be obtained from* `Pr` *by deleting zero or more statements from* `Pr`*.*
- *If* `Pr` *halts on a particular input* `I`*, then the values of the variables in* `V` *each time statement* n *is executed in* `Pr` *is the same in* `Pr` *and* `S`*. If* `Pr` *fails to terminate normally* n *may execute more times in* S *than in* `Pr`*, but* `Pr` *and* `S` *compute the same values each time* n *is executed by* `Pr`*.*

A common way to represent procedures of a program are *control flow graphs* (CFG) [112].

Definition 7.4 (Control Flow Graph). *A control flow graph for a program* `Pr` *is a directed graph* $G = (N, E, s, f)$ *in which each node* $n \in N$ *is associated with a statement from* `Pr` *and in which edge* $e \in E$ *represents the flow of control in* `Pr`*. Two special nodes are distinguished,* s *is the initial node, and* f *is the final node, to represent the beginning and the end of* `Pr` *respectively.*

Computing a slice involves identifying assignments that can affect the values of the variables given in the slicing criterion. To do this, one computes information similar to reaching definitions. This requires keeping track of the variables referenced and defined at each node in the CFG [112].

Definition 7.5 (References and Definitions).

- *Let* $REF(n)$ *be the set of variables referenced at node* n*.*

- *Let $DEF(n)$ be the set of variables defined (assigned) at node n (always a singleton set or an emptyset).*

A *program dependence graph* (PDG) [112] is a transformation of a CFG, where the control flow edges have been removed and two other kinds of edges have been inserted: *control dependence* and *data dependence* edges.

Definition 7.6 (Data Dependence). *A node j is called data dependent on node i if*

- *there is path P from i to j in the CFG: $i \xrightarrow{*} j$*
- *there is a variable v, with $v \in DEF(i)$ and $v \in REF(j)$*
- *for all nodes $k \neq i$ of path $P \Rightarrow v \notin DEF(k)$*
  (the path from i to j has no intervening definitions of v)

Control dependencies are usually defined in terms of *postdominance*. Node j is called a *postdominator* of node i, if any path from i to f must go through j. A node i is called a *predominator* of j, if any path from s must go through i. In typical programs, statements in loop bodies are predominated by the loop entry and postdominated by the loop exit.

Definition 7.7 (Control Dependence). *A node j is called (direct) control dependent on node i if*

- *there is a path P from i to j in the CFG: $i \xrightarrow{*} j$*
- *j is a postdominator for every node in P, except i*

The PDG consists of the nodes of the CFG and control dependence edges $i \xrightarrow{cd} j$ for nodes j which are control dependent on nodes i, and data dependence edges $i \xrightarrow{dd} j$ for nodes j which are data dependent on nodes i. The set of all dependencies induce a partial ordering on the statements in the program that must be followed in order to preserve the semantics of the original program.

Example 7.3. *As an example, let us return to Code Listing 7.1. Node 4 is data dependent on node 1 because: (i) node 1 defines variable* `i`*, (ii) node 4 references variable* `i`*, (iii) and there exists a path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ without intervening definitions of* `i`*. Node 5 is control dependent on node 3 because there exists a path $3 \rightarrow 4 \rightarrow 5$ such that: (i) node 4 is postdominated by node 5, (ii) node 3 is not postdominated by node 5.*

Using a program dependence graph, program slicing can be seen as *a graph reachability problem*. The slice of a dependence graph with respect to node n of the graph, is the graph that contains all nodes that (directly or indirectly) reach n via a data dependence or control dependence[2].

Algorithm 7.1 A Backwards Slicing Algorithm

```
Slice(node: Node) {
IF node is not marked THEN
  mark node as visited
  FOR all nodes n on which node depends DO
   Slice(n)
}
```

Control and data dependencies were proposed and studied for sequential programs. However, they are inadequate for representing the complete behavior of a concurrent program. In concurrent programs that share variables another type of dependence arises: *interference*. An interference dependence is a generalization of the notion of data dependence resulting from the definition and the use of variables that are common to parallel executing statements [65]. Thus, interference dependencies can only arise when data is shared between two threads.

Definition 7.8 (Interference Dependence). *Node i is interference-dependent on node j if*

- *i and j are in different threads, and*
- *there is a variable v, with $v \in DEF(j)$ and $v \in REF(i)$*

7.3 Curbing the State Explosion Problem

It should already be clarified that temporal logic model checking is a state space method suitable for the automatic analysis and the verification of the behavior of several kinds of systems. In its basic form, each model checker creates a structure that consists of all states that a system can reach, and all transitions that the system can make between those states. This structure is called the *state space*, and of course, the state space can be automatically created.

[2] In [112] a lot of different ways to compute slices are discussed.

However, a serious problem for the model checking approach is the *state explosion problem*: in general, the number of states in the global state graph may grow exponentially with the number of components. Due to the state explosion problem, one possible outcome of running a model checking algorithm is that it will fail to terminate due to the size of the model. Not surprisingly, statecharts suffer from the state explosion problem as well; e.g. any parallel activity causes exponential growth of the state space.

In order to deal with the state explosion problem, the size of the state space must be reduced to something that could be handled. More generally, given a logic $\mathcal{L}$ and a structure $\mathcal{M}$, we would like to find a smaller structure $\mathcal{M}'$ that satisfies exactly the same set of formulas of the logic $\mathcal{L}$ as $\mathcal{M}$. In order to accomplish this goal, we need a notion of equivalence between structures that can be efficiently computed and that guarantees that they satisfy the same set of formulas in $\mathcal{L}$.

Since UML statecharts are constructed in a hierarchical fashion, the state explosion may be avoided by simplifying the hierarchical states before computing the global state graph. One way to realize this is based on program slicing, as proposed in [125]. The slicing algorithm of [125] is based on equivalent EHA representations. The algorithm aims at a reduction of the state space before critical temporal properties on the behavior are formally verified by model checking algorithms. However, it is a static algorithm since no assumptions regarding the statechart's input ($\neq$ initial states) are made. Since a model checker is capable of verifying properties for all possible inputs, dynamically slicing statecharts is avoided.

Hereafter, the algorithm of [125] is referred to as the *WDQ-algorithm*.

7.4 The Slicing Algorithm

The slicing procedure, described in [125], uses extended hierarchical automata as underlying representations. These representations are high-level specifications and slicing them is in some aspects quite different from slicing programs.

In statechart models, variables are not the only entity we want to use as a basis for calculating a slice. Meaningful statecharts without variables are quite common during the early stage of system specification. In this case, states and transitions are the focus of our attention. As a direct consequence, traditional data, control and interference dependencies proposed and studied for sequential and concurrent programs, cannot be used anymore.

Instead, [125] extends traditional dependencies in terms of states and transitions, and these dependencies are capable to handle hierarchy, con-

currency and communication; concepts which are more difficult to analyze comparing with statements. To make the slicing algorithm efficient, [125] specifies the redefined dependencies by only considering the $(n-1)^{th}$, n^{th}, and $(n+1)^{th}$ layer of the EHA. The WDQ-algorithm respects, as it has to, the run-to-completion step semantics of statecharts.

The basic idea of the WDQ-algorithm, is to search states and transitions through the dependence relations according to a given slicing criterion. Additionally, if a state or transition of some sequential automaton belongs to the slice, then all states and transitions of this automaton are added to the slice. If a state (transition) belongs to the final slice, but does not take part in some dependence relation, then its entry/exit actions (effect) will be removed respectively. The found states and transitions, possibly affects all states and transitions in the criterion. The units of the minimized EHA are, of course, still sequential automata. The sliced EHA, obtained by the procedure, is sound and executable.

7.5 Dependences in an EHA

Dependences in ordinary program slicing are defined based upon the sets (REF and DEF) the nodes are associated with. Something similar will happen here.

Let $H = (F, E, \rho, A_0, V)$ be an EHA. Let $A = (\sigma_A, s^0_A, \delta_A, \lambda_A) \in F$ be a sequential automaton with $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ as defined in Definition 2.2. Now we consider the notation (s_i, t_i, s_{i+1}) for a transition from s_i to s_{i+1}, where $t_i \in \delta_A$ with $SRC(t_i) = s_i \in \sigma_A$ and $TGT(t_i) = s_{i+1} \in \sigma_A$.

Sets Associated to States Each state s ($s \in \mathcal{S}(A_0)$) comes equipped with the following sets:

- The set DV is the set of update variables of s; it includes all the variables which are assigned values in the actions below s (i.e. $\rho(s)$, the EHA below s), and can be defined and referenced outside the sub-EHA of s; it is the set of the output variables of state s, denoted as $s.DV$.
- The set UV consists of all the variables which are referenced in the actions below s; it is the set of the input variables for state s, denoted as $s.UV$.
- The set GE includes all the events which are generated in the actions below s, and can be used as the trigger events of the transitions outside s. Notation: $s.GE$.

- The set TE includes all the events which are generated outside s, and will be used as the trigger events of the transitions below s. Notation: $s.TE$.

There is a restriction placed on the use of events: the triggering event of a transition can only be generated in an automaton that is concurrent with this transition. The restriction is consistent with the fact that events are mainly used for synchronization.

Sets Associated to Transitions Each transition t ($t \in \mathcal{T}(A_0)$) has the following sets:

- $t.DV$ is the set of variables that are defined in $AC(t)$.
- $t.UV$ is the set of variables that are referenced in $AC(t)$.
- $t.CV$ denotes the variables referenced in $G(t)$.
- $t.GE$ collects the events generated in $AC(t)$.
- $t.TE$ is a singleton set containing $EV(t)$.

Definition 7.9. *A path $P(s_1, s_k)$ from s_1 to s_k in A is a sequence of transitions*

$$(s_1, t1, s2), \ldots, (s_i, t_i, s_{i+1}), \ldots, (s_k, t_k, s_{k+1})$$

7.5.1 Sequential Data Dependence

Sequential dependence occurs when activities are performed in series; where the output of one activity is the input of a next activity. A state or transition is *(directly) sequential data dependent* on a "previous" state or transition iff some output variables of the latter are used in the input of the other. We use the following formal definitions:

Definition 7.10 (Sequential Data Dependence, $\rightarrow_{sdd}$). *If $A \in F$ and $u, v \in \sigma_A$, $r, t \in \delta_A$,*

- *$u \rightarrow_{sdd} v$ iff there is a path $P(v, u)$: $(s_1 = v, t_1, s_2), (s_2, t_2, s_3), \ldots, (s_{k-1}, t_{k-1}, s_k = u)$ such that $(v.DV \cap u.UV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1 \leq i<k} t_i.DV)) \neq \emptyset$.*
- *$u \rightarrow_{sdd} t$ iff there is a path $P(SRC(t), u)$: $(s_1, t_1 = t, s_2), (s_2, t_2, s_3), \ldots, (s_{k-1}, t_{k-1}, s_k = u)$ such that $(t.DV \cap u.UV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1<i<k} t_i.DV)) \neq \emptyset$.*

- $r \rightarrow_{sdd} v$ *iff there is a path* $P(v, TGT(r))$*:* $(s_1 = v, t_1, s_2), (s_2, t_2, s_3), \ldots, (s_{k-1}, t_{k-1} = r, s_k)$ *such that* $(v.DV \cap r.UV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1\leq i<k-1} t_i.DV)) \neq \emptyset$.
- $r \rightarrow_{sdd} t$ *iff there is a path* $P(SRC(t), TGT(r))$*:* $(s_1, t_1 = t, s_2), (s_2, t_2, s_3), \ldots, (s_{k-1}, t_{k-1} = r, s_k)$ *such that* $(t.DV \cap r.UV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1<i<k-1} t_i.DV)) \neq \emptyset$.

Obviously, sequential data dependencies between states and transitions of a single sequential automaton exist iff they share some data and iff there is a path between them without intervening definitions of the shared data.

Example 7.4. *Consider Figure 7.1, the transition from* $s2$ *to* $s1$ *is sequential data dependent on* $s3$*. The latter one is sequential data dependent on* $s1$ *in turn.*

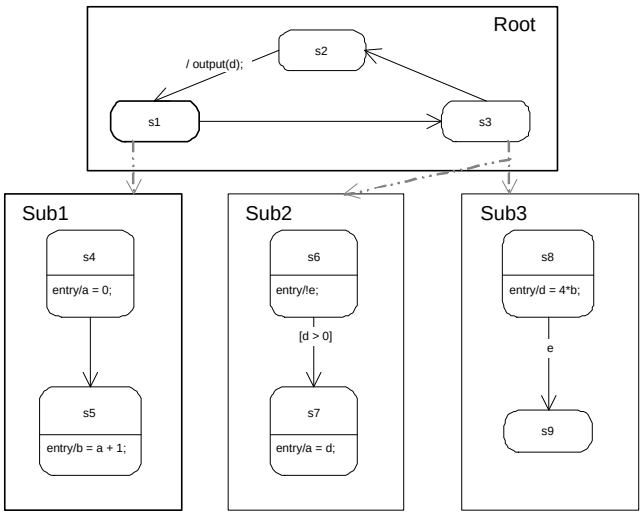


Figure 7.1: An Example EHA with Dependences

7.5.2 Parallel Data Dependence

As its name implies, a parallel data dependence can be seen as an interference dependence in the sense that such a dependence will arise when data is shared

among concurrent automata. A state or transition is *(directly) parallel data dependent* on a "concurrent" state or transition iff some output variables of the latter are used in the input of the other. We use the following formal definitions:

Definition 7.11 (Parallel Data Dependence, $\rightarrow_{pdd}$). *If $A, B \in F$ and $u \in \sigma_A, r \in \delta_A, v \in \sigma_B, t \in \delta_B$, and there are $C \in F$ and $s \in \sigma_C$ such that $A, B \in \rho(s)$, then $u \rightarrow_{pdd} v$ (or $u \rightarrow_{pdd} t$, or $r \rightarrow_{pdd} v$, or $r \rightarrow_{pdd} t$) iff $u.UV \cap v.DV \neq \emptyset$ (or $u.UV \cap t.DV \neq \emptyset$, or $r.UV \cap v.DV \neq \emptyset$, or $r.UV \cap t.DV \neq \emptyset$ respectively).*

Example 7.5. *In Figure 7.1, it is clear that $s7$ is parallel data dependent on $s8$.*

7.5.3 Synchronization Dependence

This dependence is a special interference dependence; it occurs when events are shared between concurrent automata (one automaton generates the event, the other one consumes the event as the trigger of some transition). A state or transition is *(directly) synchronization dependent* on a "concurrent" state or transition iff some events generated by the latter are used as trigger events of the other. We use the following formal definitions:

Definition 7.12 (Synchronization Dependence, $\rightarrow_{sd}$). *If $A, B \in F$ and $u \in \sigma_A, r \in \delta_A, v \in \sigma_B, t \in \delta_B$, and there are $C \in F$ and $s \in \sigma_C$ such that $A, B \in \rho(s)$, then $u \rightarrow_{sd} v$ (or $u \rightarrow_{sd} t$, or $r \rightarrow_{sd} v$, or $r \rightarrow_{sd} t$) iff $u.TE \cap v.GE \neq \emptyset$ (or $u.TE \cap t.GE \neq \emptyset$, or $r.TE \cap v.GE \neq \emptyset$, or $r.TE \cap t.GE \neq \emptyset$ respectively).*

Example 7.6. *In Figure 7.1, the transition from $s8$ to $s9$ is synchronization dependent on $s6$.*

7.5.4 Transition Control Dependence

Control dependence captures the notion that a state or a transition may affect traversal of an arbitrary transition. It refers to the flow of control inside hierarchical automata. A state or transition is *(directly) transition control dependent* on a "previous/concurrent" state or transition iff some output variables of the latter are used in the guard of the other. We use the following formal definitions:

Definition 7.13 (Transition Control Dependence, $\rightarrow_{tcd}$). *Let $A \in F$ and $r \in \delta_A$,*

- *For* $v \in \sigma_A$, $r \rightarrow_{tcd} v$ *iff there is a path* $P(v, TGT(r))$: $(s_1 = v, t_1, s_2)$, (s_2, t_2, s_3),..., $(s_{k-1}, t_{k-1} = r, s_k)$ *such that* $(v.DV \cap r.CV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1\leq i<k-1} t_i.DV)) \neq \emptyset$.
- *For* $t \in \delta_A$, $r \rightarrow_{tcd} t$ *iff there is a path* $P(SRC(t), TGT(r))$: $(s_1, t_1 = t, s_2)$, (s_2, t_2, s_3),..., $(s_{k-1}, t_{k-1} = r, s_k)$ *such that* $(t.DV \cap r.CV) \setminus ((\bigcup_{1<i<k} s_i.DV) \cup (\bigcup_{1<i<k-1} t_i.DV)) \neq \emptyset$.
- *If* $B, C \in F$ *and* $s \in \sigma_C$ *satisfy* $A, B \in \rho(s)$, $v \in \sigma_B$, $t \in \delta_B$, *then* $r \rightarrow_{tcd} v$ *(or* $r \rightarrow_{tcd} t$*) iff* $r.CV \cap v.DV \neq \emptyset$ *(or* $r.CV \cap t.DV \neq \emptyset$ *respectively).*

Example 7.7. *In Figure 7.1, the transition from* $s6$ *to* $s7$ *is transition control dependent on* $s8$.

7.5.5 Refinement Data and Control Dependence

The reason to have these two (direct) dependencies will be clarified in Section 7.6.3. It is mainly used for descending in a hierarchical automaton or to go upwards in a hierarchical automaton.

Definition 7.14 (Refinement Data Dependence, $\rightarrow_{rdd}$). *If* $A \in F$ *and* $u \in \sigma_A$, $B \in \rho(u)$, $v \in \sigma_B$, $t \in \delta_B$,

- $u \rightarrow_{rdd} v$ *iff* $(v.DV \cap u.DV) \neq \emptyset$, *or* $(u.GE \cap v.GE) \neq \emptyset$.
- $u \rightarrow_{rdd} t$ *iff* $(t.DV \cap u.DV) \neq \emptyset$, *or* $(u.GE \cap t.GE) \neq \emptyset$.

Definition 7.15 (Refinement Control Dependence, $\rightarrow_{rcd}$). *If* $A, B \in F$, $u \in \sigma_A$, $v \in \sigma_B$, *and* v *is the initial state of* B, *then* $v \rightarrow_{rcd} u$ *iff* $B \in \rho(u)$.

Example 7.8. *In Figure 7.1,* $s3 \rightarrow_{rdd} s8$, *and* $s4 \rightarrow_{rcd} s1$.

7.5.6 Dependence Relation

The dependence relation ($\rightarrow_d$) defined for hierarchical automata, is the union of the dependence relations $\rightarrow_{sdd}$, $\rightarrow_{pdd}$, $\rightarrow_{sd}$, $\rightarrow_{tcd}$, $\rightarrow_{rdd}$, and $\rightarrow_{rcd}$.

7.6 Computation of an EHA Slice

In [51], Hatcliff et al. use slicing to extract a model from the source code that can be used in verifying a given temporal property. So the property they want to verify determines the resulting "slice". They say:

> *If slicing removes variables from the system that do not influence the behavior to be checked then the model checker will run faster regardless of the particular implementation techniques it employs.*

To make our presentation of the WDQ-algorithm more concrete, let us consider model checking of statechart specifications written in linear temporal logic. When specifying properties of state machines, LTL formulae are used to reason about execution in terms of states, as well as values of particular class attributes. Thus, compared with ordinary program slicing, the temporal properties contain more information that need to be considered to construct a reduced model. As a consequence, [125] has adapted the slicing criterion to concentrate on multiple states and transitions.

Definition 7.16 (Slicing Criterion). *A slicing criterion of H is a tuple $< \{s_1, \ldots, s_k\}, \{t_1, \ldots, t_n\} >$ where $s_i \in \mathcal{S}(A_0)$, $t_i \in \mathcal{T}(A_0)$, and $s_i \neq s_j$ $(1 \leq i, j \leq k, i \neq j)$, $t_i \neq t_j$ $(1 \leq i, j \leq n, i \neq j)$.*

The WDQ-algorithm, presented in Algorithms 7.2-7.6, computes the slice of an EHA with respect to the slicing criterion according to the dependence relations in the EHA. Chapter 10 applies in great detail the slicing algorithm to the example of the coffee vending machine (see Section 3.3).

7.6.1 Some Useful Sets

The algorithm computes the reduced model using several sets:

- RS and RT keep track of the states and the transitions respectively to be reserved in the final slice.
- ES and ET contain the states and the transitions respectively which are not included in NS and NT but belong to the automata which include the newly found elements in NS and NT.
- $Refine_R$ is a boolean function used to determine whether the refinement information (e.g. actions) of the reserved elements should be kept in the final slice.
- IS and IT contain the states and the transitions respectively used to find the elements on which they depend.
- NS and NT collect the states and the transitions respectively found in each iteration; on which IS and IT depend.

7.6.2 The First Step

At step 1 (Algorithm 7.2), only the states and the transitions of the slicing criterion are included in RS and RT respectively. Assigning $True$ to their $Refine_R$ means that their refined elements should be reserved.

Algorithm 7.2 Step 1 of the WDQ-algorithm

1. Given the criterion $< \{s_1, \ldots, s_k\}, \{t_1, \ldots, t_n\} >$. Let

$$RS = \{s_1, \ldots, s_k\} \quad Refine_R(s) = True \ (\forall s \in RS)$$
$$RT = \{t_1, \ldots, t_n\} \quad Refine_R(t) = True \ (\forall t \in RT)$$

$$ES = \{s \mid \exists A \in F, ((\exists u \in RS, u \in \sigma_A) \vee (\exists t \in NT, t \in \delta_A)) \wedge (s \in \sigma_A \wedge s \notin RS)\}$$
$$ET = \{r \mid \exists A \in F, ((\exists u \in RS, u \in \sigma_A) \vee (\exists t \in NT, t \in \delta_A)) \wedge (r \in \delta_A \wedge r \notin RT)\}$$

$$Refine_R(s) = False \ (s \in ES)$$
$$Refine_R(t) = False \ (t \in ET)$$

$$RS = RS \cup ES;\ RT = RT \cup ET;\ IS = RS;\ IT = RT.$$

To guarantee that the resulting slice is sound, we have to add to ES and ET respectively those elements which are in the same sequential automata with the elements in RS and RT. That way, the execution trace inside sequential automata remains the same; meaning that the automata of the sliced EHA behave the same as the original ones, with respect to the variables of interest. Of course, at this point, we do not yet know whether the elements of the slicing criterion depend on these elements; and therefore, the value of their $Refine_R$ is falsified.

Example 7.9. *Consider Figure 7.1 again. If* $s7$ *belongs to* RS*, then it is necessary to add the other elements of the automaton* $SUB2$ *to* ES *and* ET*, because only then, the reachability of* $s7$ *is guaranteed in the sliced version of the example EHA.*

7.6.3 The Second Step

The states and the transitions on which the existing states and transitions in IS and IT depend, are computed in the second step (Algorithm 7.3). As an

example, consider the calculation of NS. The meaning of $u \rightarrow_d s$ is obvious; alle states that influences u in some way are added to the final slice.

Algorithm 7.3 Step 2 of the WDQ-algorithm

2. Compute the states and transitions which the existing states and transitions in IS and IT are dependent on by the following equations:

$$\begin{aligned} NS = &\{s \mid u \rightarrow_d s, u \in IS \wedge Refine_R(u)\} \cup \\ &\{s \mid u \rightarrow_{rcd} s, u \in IS \wedge \neg Refine_R(u)\} \cup \\ &\{s \mid t \rightarrow_d s, t \in IT \wedge Refine_R(t)\} \cup \\ &\{s \mid t \rightarrow_{tcd} s \vee t \rightarrow_{sd} s, t \in IT \wedge \neg Refine_R(t)\} \end{aligned}$$

$$\begin{aligned} NT = &\{r \mid u \rightarrow_d r, u \in IS \wedge Refine_R(u)\} \cup \\ &\{r \mid t \rightarrow_d r, t \in IT \wedge Refine_R(t)\} \cup \\ &\{r \mid t \rightarrow_{tcd} r \vee t \rightarrow_{sd} r, t \in IT \wedge \neg Refine_R(t)\} \end{aligned}$$

But what does $u \rightarrow_{rcd} s$ precisely means? Well, u is the initial state of an automaton $\in \rho(s)$ (Definition 7.15). Such a dependence relation will add to the slice those states that are higher in the hierarchy of the EHA. These states are important to guarantee that the reduced hierarchical automaton can still reach state u along some execution path; thus again a soundness aspect.

Example 7.10. *If $s6$ is added to the slice, when slicing the EHA of Figure 7.1, then the second step will also add state $s3$ to the slice, due to the refinement control dependence relation that exists between both states. This way, the model checker can reach state $s6$ along some path during verification.*

Another soundness satisfaction can be found in $t \rightarrow_{tcd} s \vee t \rightarrow_{sd} s$. Each transition has three elements: a triggering event, a guard, and a list of actions. Obviously, actions of transitions possibly may be removed in the final slice, but the final slice must retain both the triggering event and the guard, because they define the flow of control inside sequential automata. Therefore, it is important to add those states that either generate the triggering event of such transitions, or that define the variables of the guard.

7.6.4 The Third Step

Step 3 (Algorithm 7.4) searches the states and transitions which are not included in $(RS \cup NS)$ and $(RT \cup NT)$, but are in the same sequential automata with new elements of NS and NT. As before, the execution trace inside the reduced hierarchical automaton remains the same.

Algorithm 7.4 Step 3 of the WDQ-algorithm

3. Reconstruct ES and ET:

$$ES = \{s \mid \exists A \in F, ((\exists u \in NS, u \in \sigma_A) \vee (\exists t \in NT, t \in \delta_A)) \\ \wedge (s \in \sigma_A \wedge s \notin (RS \cup NS))\}$$
$$ET = \{r \mid \exists A \in F, ((\exists u \in NS, u \in \sigma_A) \vee (\exists t \in NT, t \in \delta_A)) \\ \wedge (r \in \delta_A \wedge r \notin (RT \cup NT)\}$$

7.6.5 The Fourth Step

At step 4 (Algorithm 7.5), the sets IS and IT are reconstructed. They will include the found elements that do not belong to RS and RT.

Algorithm 7.5 Step 4 of the WDQ-algorithm

4. Reconstruct IS and IT:

$$IS = (NS \setminus RS) \cup \\ \{s \mid s \in NS \wedge s \in RS \wedge \neg Refine_R(s)\} \cup \\ \{s \mid \exists A \in F, s \in ES \wedge s = s_A^0\}$$

$$IT = (NT \setminus RT) \cup \\ \{t \mid t \in NT \wedge t \in RT \wedge \neg Refine_R(t)\} \cup ET$$

To construct the set IS, each state that belongs to NS takes part in a particular dependence relation; some elements of IS are dependent on these states; the states of NS influence other elements that are already included in the slice. Thus, the states that belong to NS and to RS have to be included in IS if their $Refine_R$ is falsified. That way, the algorithm can search elements on which these states depend in turn. Additionally, all the initial states belonging to the extended set ES have to be included in IS as well; they are treated separately (with the refinement controle dependence) in the second step of the algorithm, as discussed previously. All the other states of ES are not included in IS because they are not part of any dependence relation.

The set IT is constructed in almost the same manner but contrarily to IS, it keeps track of all the elements of ET. This is because the second step of the algorithm finds dependencies for the triggering event and the guard of the transitions, as also discussed previously.

7.6.6 The Fifth, Sixth and Last Step

At step 5 (Algorithm 7.6, $Refine_R$ is refreshed, and both sets RS and RT are updated at step 6. If both IS and IT are empty, the algorithm will terminate, otherwise return to step 2.

Algorithm 7.6 Step 5, 6 and 7 of the WDQ-algorithm

5. $\forall s \in (NS \cup ES)$, $\forall t \in (NT \cup ET)$

$Refine_R(s) = True$ if $s \in NS$ $\quad Refine_R(t) = True$ if $t \in NT$
$Refine_R(s) = False$ if $s \notin NS$ $\quad Refine_R(t) = False$ if $t \notin NT$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; RT = RT \cup NT \cup ET$

7. If $IS = \emptyset$ and $IT = \emptyset$, then terminates; otherwise return to step 2.

After the algorithm terminates, all the sequential automata whose states and transitions are included in RS and RT compose the slice of H according to the slicing criterion. For any state s of RS, if $Refine_R(s)$ is false, then we do not consider its sub automata ($\rho(s)$) and we are allowed to remove the actions (entry, exit) of s. For a transition t of RT, if $Refine_R(t)$ is false, then we remove the action(s) of t.

It must already be clear that if a state or transition of some sequential automaton belongs to the slice, then all the states and transitions of this automaton belong to the slice. The sequential automata which have no state or transition in the slice will be removed.

7.7 Equivalence between Models

The properties to be verified are intended to be specified in LTL. Suppose now that the propositions in LTL formulae are of three forms: x *rop* c, in which x is a variable, *rop* is a relation operator ($<, >, \neq, \ldots$), and c is a constant; $@s_i$ is true when the current configuration includes the state s_i; $!e$ is true when the event e has been generated and belongs to the event queue of the current status.

As must be clear by now, slicing is used to reduce the complexity of hierarchical automata. Therefore, we are allowed to say that slicing is

a specific abstraction mechanism. One of the key aspects of the slicing methodology is to pick out a set of relevant states and transitions in correlation with an LTL specification. Thus, given an EHA H and a specification φ, it is desired to extract a slicing criterion from φ. Slicing H with respect to the criterion should yield a smaller residual EHA H_s that preserves and reflects the satisfaction of φ (*stuttering equivalent*) and has as little irrelevant information as possible.

Stuttering [70] refers to a sequence of identically labeled states along a path in a Kripke structure (Definition 7.17). Leslie Lamport has argued (see [70]) that the specification of a concurrent system should be invariant under stuttering, i.e. the specification should not distinguish between a sequence of states and any sequence that results from it by replacing an occurrence of a state by several copies of that state.

Definition 7.17 (Stuttering). *Stuttering [33] occurs in a path p of states when a state occurs two or more times consecutively; for example, if $p = s_0, s_1, s_2, \ldots$ then there is $i \geq 0$ such that $s_i = s_{i+1}$ so $p = s_0, s_1, \ldots, s_i, s_i, s_{i+2}, \ldots$*

We call a finite sequence of identically labeled states a *block*. Intuitively, two paths are stuttering equivalent (Definition 7.18) when they can be partitioned into infinitely many blocks, such that the states in the k-th block of one are labeled the same as the states in the k-th block of the other one. Note that the corresponding blocks may have different lengths.

Definition 7.18 (Stuttering Equivalence). *Two infinite paths $p = s_0, s_1, \ldots$ and $p' = r_0, r_1 \ldots$ are stuttering equivalent [33], written as $p \sim_{st} p'$ if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$*

$$\lambda(s_{i_k}) = \lambda(s_{i_k+1}) = \ldots = \lambda(s_{i_{k+1}-1})$$

$$= \lambda(r_{j_k}) = \lambda(r_{j_k+1}) = \ldots = \lambda(r_{j_{k+1}-1})$$

More meaningful to us is to define the concept of *stuttering equivalence* with respect to a property φ. Intuitively, two executions are *φ-stuttering equivalent* (Definition 7.19) if they can be defined as a concatenation of blocks such that the atomic propositions of the i-th block of both executions have the same intersection with φ, for each $i > 0$. Figure 7.2 illustrates two stuttering equivalent paths with respect to a property in which only propositions p and q occur.

Definition 7.19 (φ-Stuttering Equivalence). *Two infinite paths $p = s_0, s_1, \ldots$ and $p' = r_0, r_1 \ldots$ are stuttering equivalent with respect to a property φ [33], written as $p \sim_\varphi p'$ if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$*

$$\lambda(s_{i_k}) \cap \varphi = \lambda(s_{i_k+1}) \cap \varphi = \ldots = \lambda(s_{i_{k+1}-1}) \cap \varphi$$
$$= \lambda(r_{j_k}) \cap \varphi = \lambda(r_{j_k+1}) \cap \varphi = \ldots = \lambda(r_{j_{k+1}-1}) \cap \varphi$$

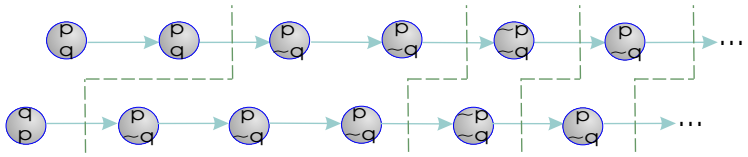


Figure 7.2: Two stuttering equivalent paths

The definition of φ-stuttering equivalence can be extended to structures. This roughly means that on the property φ both structures have an equivalent behavior.

Definition 7.20 (φ-Stuttering Equivalence for Structures). *The structures M and M' are φ-stuttering equivalent iff*

- *their initial states agree on φ: $\lambda(s_0) \cap \varphi = \lambda(s'_0) \cap \varphi$*
- *for each path p of M starting from s_0, there exists a path p' of M' starting from s'_0 such that $p \sim_\varphi p'$*
- *vice versa, for each path p' of M' starting from s'_0, there exists a path p of M starting from s_0 such that $p' \sim_\varphi p$*

We say that an LTL formula$f$ is *invariant under stuttering* when its interpretation remains the same under state sequences that differ only by repeated states. In other words, given any state sequence, we can repeat any of its states without changing the interpretation of f. More formally,

Definition 7.21 (Stuttering Invariance). *An LTL formula, f, is stuttering invariant [26] if and only if for each pair of paths p and p', such that $p \sim_{st} p'$,*

$$p \models f \text{ if and only if } p' \models f.$$

It is easy to see that any LTL formula f without the next ($\mathbf{X}$) operator is invariant under stuttering. For example, $\mathbf{G}f$ is invariant under stuttering

because no matter how much we repeat states, we cannot change the value of f. The **G** operator says that in every state some proposition must be true, a repeated state will not affect the truth value of a formula quantified with always. The same can be said of the temporal operators **F**, and **U**. Clearly, formulas using only these operators are stutter invariant because they are not affected by repeating states. However, the **X** operator says that in whatever state occurs after a single transition, some proposition must hold. If a state is repeated, the next operator could be evaluated on the repeated state and not on the actual next state in the system. Thus, not all formulas using the next operator are safe to use because this could cause incorrect verification results if a formula is used that can be affected by stuttering [33]. This is illustrated in Example 7.11.

Example 7.11. *To understand how stuttering can change the interpretation of* $\boldsymbol{X}f$*, let us consider two paths* $p = (s_0, s_1, s_2, s_3, \ldots)$ *and* $p = (s_0, s_0, s_1, s_2, s_3, s_3, \ldots)$ *that are stuttering equivalent. Only path* p' *has stuttering states. Suppose now that* $s_0(f)$ *is* $false$ *and that* $s_1(f)$ *is true. It is easy to understand that* $p \models \boldsymbol{X}f$*, while* $p' \not\models \boldsymbol{X}f$*.*

Lemma 7.1 states that the satisfaction of an LTL_X formula[3] φ for two φ-stuttering equivalent structures is the same [26]. In other words, if M and M' are two stuttering equivalent transitions systems, then M satisfies a given LTL_X specification if and only if M' also does. Thus, if the property is written in LTL_X then it does not matter whether we check a model M against this property or an equivalent model M'. Moreover, Peled and Wilke have shown that stutter-invariant properties are expressible without the next-time operator [101].

Lemma 7.1. *Given a* LTL_X *formula* φ*, if two structures* M *and* M' *are* φ*-stuttering equivalent, i.e.* $M \sim_\varphi M'$*, then* $M, s_0 \models \varphi$ *iff* $M', s_0 \models \varphi$*.*

We can extract the slicing criterion (taken from [125]) for a formula φ and guarantee that the transition system of the sliced EHA is φ-stuttering equivalent to the original one; i.e. transitions that are not φ-stuttering are preserved in the final slice. For variable propositions $x \; rop \; c$, only definitions of the variable x may cause the variable to change value. This suggests that for each proposition $x \; rop \; c$ in a given property φ, each state or transition that assigns value to x should be included in the residual slice. For a proposition $@s_i$, it is obvious that s_i should be included in the slice. For the proposition $!e$, the states and the transitions whose actions generate event e should be included in the slicing criterion.

[3] LTL_X is the subset of LTL formulae without the next time operator appearance.

Definition 7.22 (φ-Criterion). *Given a LTL_X formula φ over an EHA H, V and E are the sets of all variables and events occurring in φ respectively. Define φ-criterion $< \{s_1, \ldots, s_k\}, \{t_1, \ldots, t_n\} >$, and $s_i \neq s_j$ $(1 \leq i, j \leq k, i \neq j)$, $t_i \neq t_j$ $(1 \leq i, j \leq n, i \neq j)$, where*

- *$\{s_1, \ldots, s_k\}$ is the set of all states which assign values to variables in V or generate the events in E plus the set of all states appearing in state propositions of φ;*
- *$\{t_1, \ldots, t_n\}$ is composed of the transitions whose actions contain assignments to variables in V or generations of events in E.*

Let us finish with the following theorem (taken from [125]) that guarantees that the transition system of the sliced EHA is φ-stuttering equivalent to the transition system of the original EHA.

Theorem 7.1. *Given an EHA H and an LTL_X formula φ. Let H_s be the result of slicing H with respect to the slicing criterion 7.22, and M and M' the labeled transition systems of H and H_s respectively. Then, $M \sim_\varphi M'$ holds.*

What about CTL?

As we know from Chapter 5, the model for the temporal logic CTL is a branching structure. Without the next-time operator, two structures can have corresponding stuttering equivalent sequences but still be distinguished as they have different branching points. Thus for CTL_X, we require that the slicing procedure generates a reduced state space that is *stuttering bisimilar* to the full original state space. This way, two structures will satisfy the same formulae expressible in CTL_X.

A *stuttering bisimulation* (Definition 7.23) relates states from two Kripke structures [22]. Initial states are related, and related states are labeled with the same proposition symbols. If two states are related and from one state a transition is possible, then it should be possible to simulate this transition from the related state, after first doing zero or more stuttering transitions, i.e., transitions that do not change the labeling.

Definition 7.23 (Stuttering Bisimulation). *A stuttering bisimulation between Kripke structures $(\sigma, \sigma_0, \delta, \lambda)$ and $(\sigma', \sigma'_0, \delta', \lambda')$ is a relation $R \subseteq \sigma \times \sigma'$ such that*

- *$(\sigma_0, \sigma'_0) \in R$*
- *If $(r, s) \in R$ then $\lambda(r) = \lambda'(s)$*

- *If* $(r, s) \in R$ *and* $\delta(r, r')$ *then there exists for some* $n \geq 0$, $s_0, s_1, \ldots s_n$ *such that* $s_0 = s$ *and for all* $i < n$, $\delta'(s_i, s_{i+1})$, $(r, s_i) \in R$ *and* $(r', s_n) \in R$
- *If* $(r, s) \in R$ *and* $\delta'(s, s')$ *then there exists for some* $n \geq 0$, $r_0, r_1, \ldots r_n$ *such that* $r_0 = r$ *and for all* $i < n$, $\delta(r_i, r_{i+1})$, $(r_i, s) \in R$ *and* $(r_n, s') \in R$

7.8 Methodology Extended

As shown by Figure 7.3, our tool is extended with a slicing assistant. This assistant takes as input both an EHA and a slicing criterion and computes a sliced EHA. The WDQ-algorithm is the underlying algorithm of the slicing assistant. Chapter 10 applies in great detail the slicing algorithm to the example of the coffee vending machine (see Section 3.3).

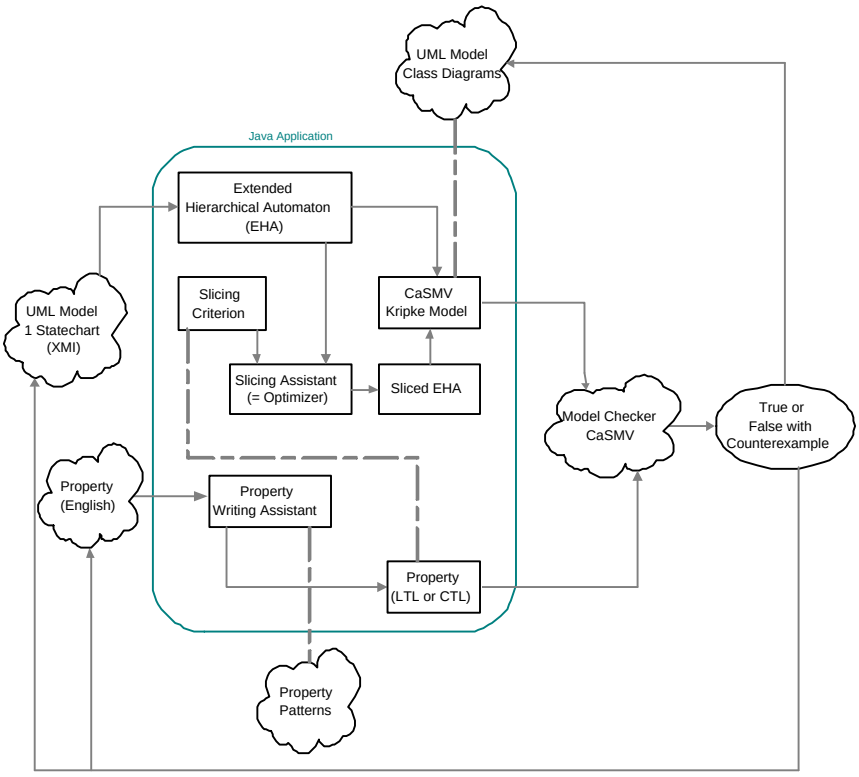


Figure 7.3: Tool Architecture

7.9 Conclusions and Related Work

When inspecting extended hierarchical automata, the first difficulty is usually to overcome the complex inner structures of non-trivial automata, and to separate the important parts from the unimportant ones. Slicing can do a lot of this automatically. This reduces both the amount of work we have to do to inspect a certain property as well as reducing the chance of mistakes.

Slicing hierarchical automata, as a pre-processing step for finite-state verification, seems to be a useful method to reduce the complexity of a statechart in order to aid verification like model checking. This chapter has covered one particular algorithm to compute a static, backwards slice based on dependence analysis with respect to the property in model checking UML Statecharts. We have explicitly avoided a dynamic slicing algorithm since model checkers are able to exhaustively verify the model for all possible inputs. Chapter 10 illustrates that the slicing algorithm indeed removes some states and transitions that are irrelevant to the property to be verified. Additionally, it proves that the algorithm sometimes is not able to slice away anything.

Besides the algorithm described in this chapter, the literature contains a few other techniques to slice automata. Specifically, slicing is used to reduce the statecharts of the *Requirements State Machine* modeling language [52] in order to improve the understanding of the design specification. It describes the control dependence and data dependence informally, and it assumes that there exist no variables in the model and the actions can only generate events. This lack of variables in the model is also present in [43].

CHAPTER 8

Internal Broadcasting: As Rich As Needed

To repeat what others have said, requires education,
to challenge it, requires brains.
Mary Pettibone Poole.

When a statechart has orthogonal regions, then this semantically means that there are independent aspects of the object, each modeled by a (sub) state machine while the object is in the concurrent state. These independent aspects are allowed to communicate, but they do so in well-defined ways [37, 38]. One way that regions communicate is that one orthogonal region may create an event that is consumed by other regions. This is called *internal broadcasting* or *synchronization*.

Although some slicing algorithms have been proposed for statecharts, no slicing algorithm has been proposed which can be used to handle the problem of slicing statecharts with orthogonal regions correctly. Efficiently calculating precise slices in the presence of concurrency and communication is challenging, because it is difficult to statically reason about the dependencies that arise when orthogonal regions perform communication.

The goal of this chapter is to show how the algorithm presented in Chapter 7 can be improved to yield an algorithm that is efficient yet effective for reducing the number of *interference dependencies* used in slicing statecharts with concurrent states. The key idea of improving the precision of interference dependencies is to define a happens-before relation [69] in terms of states and transitions. This will be achieved by exploiting the internal broadcasting mechanism while respecting the statechart's execution semantics. Moreover, the size of statecharts models is reduced, which even more improves the efficiency of model checking.

8.1 Preliminary Concepts

This section provides necessary background knowledge used to optimize the WDQ-algorithm; to improve the precision of the interference dependencies.

8.1.1 The Broadcasting Mechanism

In UML each class has an optional statechart which describes the dynamic behavior of all instances of the class [96]. As a consequence of integrating statecharts into the object-oriented paradigm, UML statecharts have many extensions caused by the need of modeling communication between objects. These are, for example, several different types of events modeling (a)synchronous message processing, timing, user interaction and other properties of object communication.

Note that not all classes have a state model. Typically, only objects with detailed state-based behavior will require the creation of a state diagram. At any moment in time, a state machine is in one state. If this appears as a limitation for a particular object, then we can define this object as consisting of various parts, each of them defined using a separate state machine. At this point, the state diagram uses at least two orthogonal regions to represent related but independent states that may be active concurrently with other states.

The independent states are allowed to communicate in only a few well-defined ways [37, 38]. For example, when an object receives an event from the environment, it is received by all of its active orthogonal components. That means that all those active regions may (or may not) act upon it. This is called *broadcasting* or *multicasting* of events. As another example, communication between orthogonal regions can also be achieved using *internal broadcasting* or *synchronization*, i.e. events generated in one orthogonal region are broadcasted to the other orthogonal regions, thus providing a communication mechanism between the regions.

The order that the orthogonal regions handle a broadcast event is non-deterministic.

Example 8.1. *Figure 8.1 shows an example of both broadcast communication types. If the object receives an event* `g`*, it is logically sent to all active orthogonal regions. Of course, the event need not be acted on in all regions. For example, if all regions are in their initial states when the object receives the event* `g`*, what happens? Obviously, region* `A` *discards the event, region* `B` *moves to substate* `B1`*, and region* `C` *transitions to substate* `C1`*. When transition* `bt2` *is taken in region* `B`*, it sends the event* `e`*. This event is sent*

to all active regions of the object. In this case, it may cause only a transition in region `A`*; the other regions discard the event.*

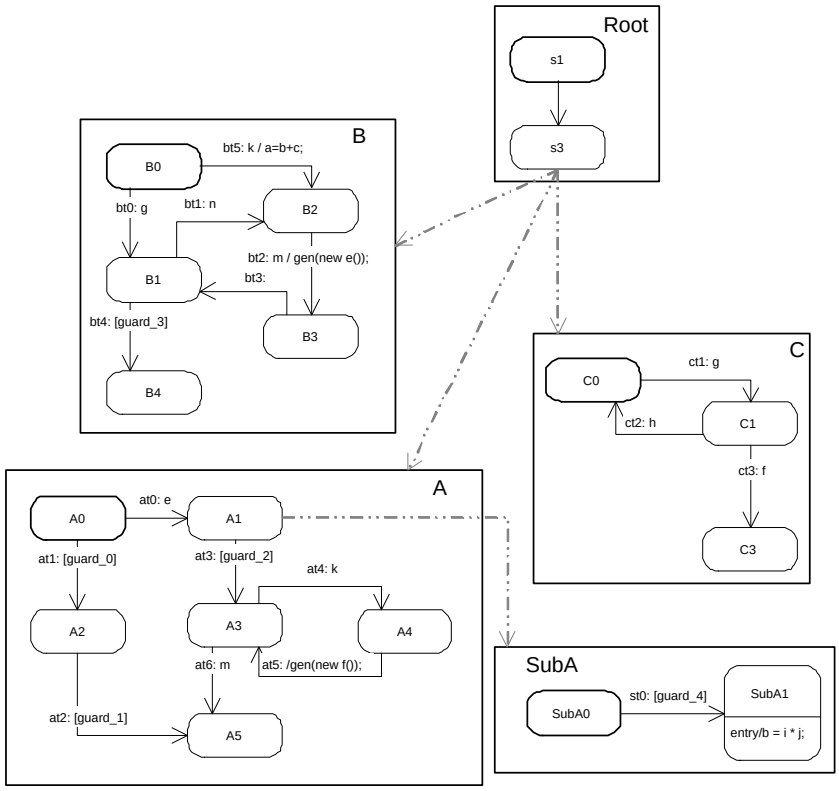


Figure 8.1: An Example of Broadcasting

8.1.2 Lamport's Happens-Before Relation

The happens-before relation defined by Lamport in [69] is the relation of causal ordering. With such a relation we are capable to grasp the concept of one event happening before another in a distributed system. The relation is defined as follows:

Definition 8.1 (happens-before ($\rightarrow_{hb}$)). *The relation $\rightarrow_{hb}$ on the set of events of a system is the smallest relation satisfying the following two conditions:*

1. *If a and b are events in the same process, and a comes for b, then $a \rightarrow_{hb} b$. (irreflexive)*
2. *If a is the sending of a message by one process, and b is the receipt of the same message by another process, then $a \rightarrow_{hb} b$. (antisymmetric)*

The first condition is simply the sequential order ($\overset{S}{\rightarrow}_{hb}$) of events [64], which also refers to the sequential flow of control inside the same process. The second condition describes the relation between corresponding events on different processes; also called the concurrent order ($\overset{C}{\rightarrow}_{hb}$) of events [64], which is established by communication and synchronization between parallel processes.

Example 8.2. *Figure 8.1 illustrates that the generation of the event* `e` *in the action list of transition* `bt2` *happens before the same event is consumed to trigger transition* `at0`*. Clearly, the generation of an event is equivalent to the sending of a message, while the consumption of an event is the same as the reception of a message. This means that the internal broadcasting mechanism (Section 8.1.1) implies a happens-before relation between states and transitions of different orthogonal regions (see further in Section 8.3). Here, $bt2 \rightarrow_{hb} at0$.*

Another requirement of the happens-before relation is transitivity:

$$\text{if } a \rightarrow_{hb} b \text{ and } b \rightarrow_{hb} c \text{ then } a \rightarrow_{hb} c$$

Consequently, the happens-before relation may also be defined as the transitive, irreflexive closure of the union of the two relations $\overset{S}{\rightarrow}_{hb}$ and $\overset{C}{\rightarrow}_{hb}$ as follows [64]:

$$\rightarrow_{hb} = (\overset{S}{\rightarrow}_{hb} \cup \overset{C}{\rightarrow}_{hb})^{+}$$

8.1.3 Digraph Definitions

When a graph is called a *directed graph* [130], it is usually abbreviated to *digraph*. A digraph is an ordered pair $G = (\sigma, \delta)$ with σ a set of "nodes" or "vertices", and δ a set of ordered pairs of vertices (a, b), with $a, b \in \sigma$, called "directed edges", "arcs" or "arrows". In some sense, UML statecharts are directed graphs.

Strongly Connected We say that vertex `x` is *reachable* from `y` if there is a directed path from `y` to `x`. Two vertices are *strongly connected* [130] if each is reachable from the other, and a digraph is called strongly connected if every vertex is strongly connected to every other vertex.

Strongly Connected Component A *strongly connected component* [130] is a maximal subset of vertices of a digraph and edges between them that forms a strongly connected graph. It is a maximal subgraph in which every vertex is reachable from every other vertex.

8.2 The WDQ-algorithm's Inefficiency

It must already be clear that slicing is becoming increasingly popular as useful preprocessing step in the construction of finite state models for automated verification like model checking. Slicing not only reduces the state space to improve the efficiency of model checking, but at the same time, slicing also improves the understanding of the design specification. Naturally, the aim of all slicing algorithms is to compute slices that are as minimal as possible, and that are as precise as possible. What do we exactly mean with precise slices?

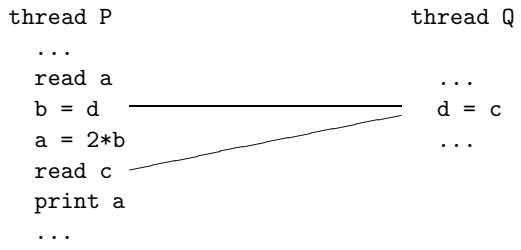
8.2.1 Imprecise Slices

To grasp the reason for imprecise slices (and thus also for precise slices), let us first move to traditional program slicing of concurrent programs. Code Listing 8.1 shows two threads `P` and `Q` that execute in parallel (taken from [66]). This threaded program has two interference dependencies; one due to a definition and a usage of variable `d` (`b=d`→`d=c`), the other one due to the access to variable `c` (`d=c`→`read c`). If, during slicing, both dependencies are added to the slice, then the slice is made imprecise. The reason for the imprecision is that there is no possible execution where the `read c` statement has an influence on the assignment `b=d`, i.e. the latter statement is always executed before the `read c` statement. Summarized, when calculating a slice, it is important to consider the execution chronology to avoid that unrealistic dependencies (thus irrelevant information) are added to the final slice.

Why is it that the WDQ-algorithm (Chapter 7), which downsizes the state space of extended hierarchical automata considerably, returns imprecise slices at the moment statecharts have concurrency features? The reason for this is quite similar as for concurrent programs, i.e. the algorithm adds interference dependencies to the slice which cannot happen in real executions. As a direct consequence, too much states, transitions and their corresponding actions may be included in the final slice.

As an example, consider again the automaton in Figure 8.1. This concurrent automaton has one parallel data dependence ($bt5 \rightarrow_{pdd} A1$). This is because a substate of `A1` defines the variable `b` used in the action

Code Listing 8.1 A Two-Threaded Program



list of transition `bt5`. Nothing more, nothing less. If during slicing, this dependence is added to the slice, then the slice is made imprecise, i.e. the whole automaton `A1` (thus $\rho(A1)$) will belong to the final slice. The reason for the imprecision (and consequently the addition of irrelevant information) is that the mentioned parallel data dependence can never happen in real execution because of the following facts:

- Following the sequential flow of control inside region `A`, we know that transition `at0` executes before state `A1` can be properly entered.
- Transition `bt2` executes before transition `at0` (Example 8.2).
- Following again the sequential flow of control inside region `B`, we know that transition `bt5` executes before transition `bt2`.
- To conclude, it is impossible that transition `bt5` uses the value of variable `b` defined in a substate of state `A1`. Indeed, transition `bt5` always executes before transition `at0`, thus before state `A1` has been entered.

Here, the reason the WDQ-algorithm returns imprecise slices is completely due to the definitions of the interference dependencies. These are defined in a rather general way in the sense that only the sharing of variables is considered without bothering the execution chronology between states and transitions. Towards verification, this simply means that the size of the state space is not reduced as much as possible. If, however, such interference dependencies take happens-before orderings into account, then, for our example, the adding of the automaton `SubA` is avoided.

8.2.2 Towards More Precise Slices

The goal in slicing statecharts is to produce a more "precise" slice. This is a slice that more closely reflects the relevant paths in a statechart. The

"preciser" a slice is, the smaller the resulting state space, and the faster a model checker can verify the correctness of a given temporal formula. Chapter 11 illustrates this.

From the previous section we learn that more precise slices can be obtained by restricting the interference dependencies in terms of a happens-before relation. As an example, if a happens before b, then a never can be dependent on b. Placing more conditions on the existence of such dependencies makes them less imprecise. As we will see later, the internal broadcasting mechanism plays a mayor role here.

At this point, it is important to realize that the dynamic execution semantics of statecharts is causal. Once a transition is fired, it cannot be interrupted and each RTC-step possibly consists of a series of micro-steps before a stable state configuration is reached. As mentioned in Section 7.5, to realize synchronization, the WDQ-algorithm places a restriction on the use of events inside statecharts; it is assumed that the triggering event of a transition can only be generated in the automata that are concurrent with this transition. That way, each synchronization dependence, which refers to synchronization points, dictates the execution chronology between states and transitions.

8.3 A Graph-Theoretic Approach

Clearly, to make the WDQ-slicing-algorithm more precise, it is necessary to determine the order in which two events (i.e. states/transitions) have occurred. For concurrent systems it is sometimes impossible to say which of the two events has occurred first. Lamport's happens-before relation (Definition 8.1) is used in obtaining an ordering of events in a broad range of systems.

To formally define a happens-before relation on statecharts, denoted by $\rightarrow_{so}$, we first define two other relations (Sections 8.3.2-8.3.3): statechart sequential order or $\overset{S}{\rightarrow}_{so}$ and statechart concurrent order or $\overset{C}{\rightarrow}_{so}$. The transitivity of the order relation $\rightarrow_{so}$ has some interesting properties, as discussed in Section 8.3.4. The statechart happens before relation $\rightarrow_{so}$ is then the irreflexive transitive closure of the two relations $\overset{S}{\rightarrow}_{so}$ and $\overset{C}{\rightarrow}_{so}$:

$$\rightarrow_{so} = (\overset{S}{\rightarrow}_{so} \cup \overset{C}{\rightarrow}_{so})^{+}$$

Combining the internal broadcasting mechanism with causality gives us a very effective way to construct $\overset{C}{\rightarrow}_{so}$. That means that all possible synchronizations will be investigated under the background of the dynamic execution

step semantics and the sequential execution semantics inside regions. But one assumption definitely is necessary: concurrent states may not have multiple generations of the same event. Finding preserved dependencies becomes otherwise undecidable since, statically, it cannot be decided which sending of the message causes the triggering of a particular transition since this requires the availability of dynamic information. Additionally, to construct $\overset{C}{\rightarrow}_{so}$, the notion of a "predecessor" and a "successor" shall be thoroughly used. The notions are defined using graph-theoretical concepts. But first, we define some definitions and concepts, which are extensively used in the following sections.

Useful Definitions or Concepts

Let $H = (F, E, \rho, A_0, V)$ be an EHA. Let $A = (\sigma_A, s_A^0, \delta_A, \lambda_A) \in F$ be a sequential automaton with $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ as defined in Definition 2.2. The corresponding **digraph of automaton A** consists of a set of nodes being σ_A, and a set of directed edges being δ_A.

Definition 8.2 (Mutilated Automaton of A Ending in x (MAE_x)). *Let $A \in F$ and $x \in (\sigma_A \cup \delta_A)$. The mutilated automaton of A ending in x, denoted as MAE_x, is the maximal (near)[1] sub-automaton of A ending in x (x included), i.e. x can be reached from each state and each transition of MAE_x.*

Example 8.3. *Let us consider region B of Figure 8.1. As visualized in Figure 8.2, the mutilated automaton of B ending in transition $bt4$ (MAE_{bt4}) consists of all states and transitions of B, except from state $B4$. Similarly, transition $bt4$ and state $B4$ is not contained in MAE_{bt2}, as shown in Figure 8.3.*

Definition 8.3 (Mutilated Automaton of A Starting in x (MAS_x)). *Let $A \in F$ and $x \in (\sigma_A \cup \delta_A)$. The mutilated automaton of A starting in x, denoted as MAS_x, is the maximal (near)[1] sub-automaton of A starting in x (x included), i.e. from x we can reach each state and each transition belonging to MAS_x.*

Example 8.4. *Let us consider region A of Figure 8.1. As visualized in Figure 8.4, the mutilated automaton of B starting in $at0$ (MAS_{at0}) does not contain the states $A0$ and $A2$, and does not contain the transitions $at1$ and $at2$, simply because we cannot reach them from transition $at0$.*

[1]Note that MAE_x and MAS_x are not complete automata if x belongs to δ_A.

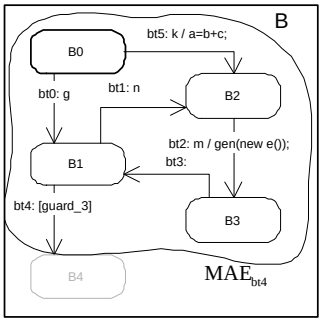


Figure 8.2: MAE_{bt4}

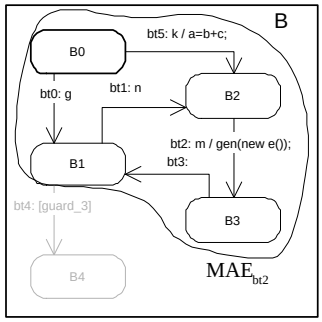


Figure 8.3: MAE_{bt2}

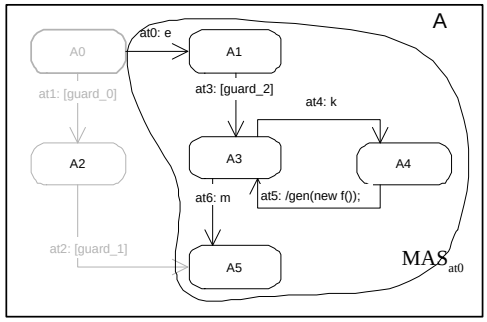


Figure 8.4: MAS_{at0}

8.3.1 Predecessors and Successors

Everyone knows that a *predecessor* is any activity that must be completed before a specified activity can begin. Everyone also knows that a *successor* is any activity whose start depends on the finish of a predecessor activity.

Example 8.5. *As can be seen on Figure 8.5, transition* `bt0` *is a predecessor of state* `B4`*. Obviously, the latter one is successor of the first. State* `B3` *is at the same time a predecessor and a successor of state* `B2` *because these states are strongly connected.*

Example 8.5 illustrates that the elements (being states and/or transitions) of an automaton can either be predecessors or successors of other elements. Sometimes they even play the role of both. This all depends on the way the statechart is executed; i.e. the dynamic execution semantics has a significant influence.

The happens-before relation between states and transitions is constructed using predecessor and successor information extensively. To do this precisely, a restriction is placed on the meaning of predecessor and successor. A state, or a transition, is now called a predecessor (successor) of an element if it *always* executes before (after) that element along some execution path of the automaton. With *always*, we mean that no predecessor (successor) ever has the chance to be a successor (predecessor) as well, no matter how an automaton transitions between states. Definitions 8.4-8.5 formally define the set of predecessors and successors of an element respectively.

Definition 8.4 (Predecessors of an Element ($pred_{element}$)). *Let $A \in F$ and $x \in (\sigma_A \cup \delta_A)$. Let MAE_x be the mutilated automaton of A ending in x. Let SCC_x ($\subseteq MAE_x$) be a strongly connected component of A containing x. Then, the set of all predecessors of element x is defined as follows:*

$$pred_x := \begin{cases} MAE_x \setminus \{x\} & \text{if } SCC_x = \emptyset \\ MAE_x \setminus SCC_x & \text{otherwise} \end{cases}$$

Example 8.6. *For all possible executions of the orthogonal region `B`, the predecessors of transition `bt2` is given by the set $pred_{bt2} = \{B0, bt0, bt5\}$ (Figure 8.1/8.5). To reach transition `bt2` some elements of $pred_{bt2}$ must be executed. As clearly visualized on Figure 8.5, elements that are both predecessors and successors of bt2 form a strongly connected component with bt2 (cycle). These elements are removed from MAE_{bt2} after all.*

Definition 8.5 (Successors of an Element ($succ_{element}$)). *Let $A \in F$ and $x \in (\sigma_A \cup \delta_A)$. Let MAS_x be the mutilated automaton of A starting in x. Let MAI_x consist of MAE_x and the mutilated automaton of A containing all paths, starting in s_A^0 but not going through x. Let SCC_x ($\subseteq MAS_x$) be a strongly connected component of A containing x. Then, the set of all successors of element x is defined as follows:*

$$succ_x := \begin{cases} MAS_x \setminus \{x\} & \text{if } MAI_x = \emptyset \wedge SCC_x = \emptyset \\ (MAS_x \setminus MAI_x) \setminus \{x\} & \text{if } SCC_x = \emptyset \\ (MAS_x \setminus MAI_x) \setminus SCC_x & \text{otherwise} \end{cases}$$

Example 8.7. *For all possible executions of the orthogonal region `A`, the successors of transition `at0` is given by the set $succ_{at0} = \{A1, at3, A3, at4, A4, at5, at6\}$ (Figure 8.5). Obviously, state `A5` is omitted from this set because it is reachable from an arbitrary path that does not pass transition `at0`.*

From the definitions of predecessors and successors respectively, it follows that $\forall x \in pred_e : pred_x \subseteq pred_e$, $\forall x \in succ_r : succ_x \subseteq succ_r$ respectively.

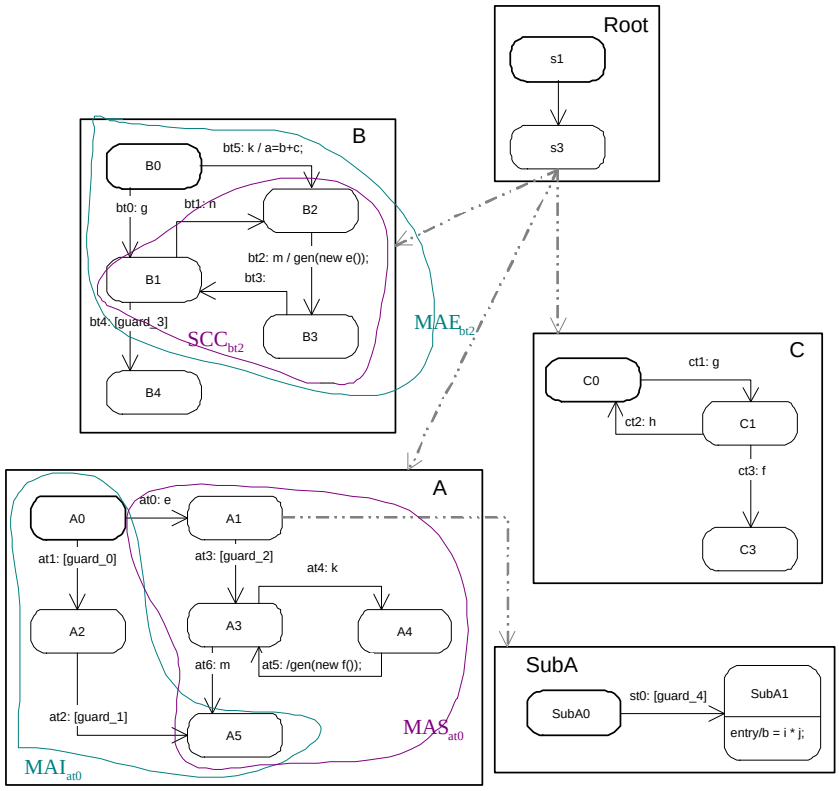


Figure 8.5: An Example of a Predecessor/Successor Set Calculation

8.3.2 Sequential Order

The sequential order comes from the sequential execution of events within the same process (Definition 8.1). In the case of statecharts, the definition of sequential ordering slightly changes. Here, states and transitions represent the events while a sequential automaton maps to a single process. The sequential order, denoted as $\xrightarrow{S}_{so}$, between states and transitions follows from the different ways a sequential automaton transitions between states.

Definition 8.6. *The relation* $\xrightarrow{S}_{so}$ *on states and transitions of an automaton is a relation satisfying the following condition: if* $A \in F, x \in (\sigma_A \cup \delta_A)$ *then*

(1) $pred_x \xrightarrow{S}_{so} x$[2]*, (2)* $x \xrightarrow{S}_{so} succ_x$[3]*, for all possible executions of A.*

Example 8.8. *In Figure 8.1, some elements of* $pred_{bt2} = \{B0, bt0, bt5\}$ *are guaranteed to execute before* `bt2`*, if they execute in a possible execution of region* `B`*. If they do not execute, the automaton never has a chance to reach* `bt2`*. Similarly, transition* `at0` *is guaranteed to execute before its successor set (*$succ_{at0} = \{A1, at3, A3, at4, A4, at5, at6\}$*).*

8.3.3 Concurrent Order

Events (states and transitions) on distinct processes (orthogonal regions) may be connected with the concurrent order relation, denoted as $\xrightarrow{C}_{so}$, if they synchronize. A synchronization point defines a property similar to Lamport's happens-before relation (Definition 8.1); it is assumed that "a message cannot be delivered before its sending".

Definition 7.12 defines four types of sychronization dependencies but only two of them are useful to define the concurrent order relation. This has anything to do with the execution semantics, as we will explain later on.

Two Orthogonal Regions

A State Synchronization Dependent on a State/Transition "A state u is synchronization dependent on another element (transition t or state v)" means that a transition inside u is triggered by an event generated by either t or v. State u is definitely a composite state, either sequential or concurrent. A dependence like this makes it impossible to specify the guaranteed execution chronology for all possible parallel executions of both automata, because the flow of control inside u is unpredictable.

A Transition Synchronization Dependent on a Transition "A transition r is synchronization dependent on a transition t" means that the triggering event of r is generated by one of the actions of t. Both transitions belong to different regions of some concurrent state.

Definition 8.7. *The relation* $\xrightarrow{C}_{so}$ *on states and transitions of concurrent automata is a relation satisfying the following condition: if* $A, B \in F, r \in \delta_A, t \in \delta_B$*, and there are* $D \in F$ *and* $s \in \sigma_D$ *such that* $A, B \in \rho(s)$ *and if* $r \rightarrow_{sd} t$ *(Definition 7.12) then* $t \xrightarrow{C}_{so} r$*, for all possible parallel executions of D.*

[2] $\forall y \in pred_x : y \xrightarrow{S}_{so} x$

[3] $\forall y \in succ_x : x \xrightarrow{S}_{so} y$

If $r \rightarrow_{sd} t$, then t is guaranteed to execute before r, if they execute in a program run, due to the dynamic execution semantics. Once t is executed, it cannot be interrupted. Therefore, transition r makes a change to be enabled for firing, if its triggering event is available in the environment (i.e. kept in the FIFO queue) and if t has finished execution. Since t is the only transition that is capable to generate the triggering event of r, it has to be executed before r.

Example 8.9. *Let us consider Figure 8.5 again. Transition* `at0` *is synchronization dependent on transition* `bt2` *because the broadcast event* `e` *(the trigger of* `at0`*) is generated in the action list of* `bt2`*. We safely conclude that* $bt2 \overset{C}{\rightarrow}_{so} at0$*. Following a similar reasoning, we may conclude that* $at5 \overset{C}{\rightarrow}_{so} ct3$*.*

A Transition Synchronization Dependent on a State "A transition r is synchronization dependent on state u" means that the triggering event of r is generated by the actions of u. Both elements belong to different substates of some concurrent state. Important to know is that u is either a simple or a composite state. If it is composite, it contains all information of its children, positioned lower in the EHA.

Definition 8.8. *The relation* $\overset{C}{\rightarrow}_{so}$ *on states and transitions of concurrent automata is a relation satisfying the following condition: if* $A, B \in F, r \in \delta_A, v \in \sigma_B$*, and there are* $D \in F$ *and* $s \in \sigma_D$ *such that* $A, B \in \rho(s)$ *and if* $r \rightarrow_{sd} v$ *(Definition 7.12) then* $pred_v \overset{C}{\rightarrow}_{so} r$*, for all possible parallel executions of* D*.*

If $r \rightarrow_{sd} v$ then v executes before r, if they execute in a program run, due to the dynamic execution semantics. Transition r only makes a change to be enabled for firing, if its triggering event is available in the environment, i.e. kept in the FIFO queue. Since v is the only state capable to generate the triggering event of r, it has to be executed before r. From this, we may conclude that $v \overset{C}{\rightarrow}_{so} r$, but this order relation is not legal for every possible execution of the concurrent automaton D. For example, if v is a simple state, the dynamic execution semantics still can activate internal activities, even after r has finished execution. As another example, if v is a composite state, some child of v is responsible for the generation of the triggering event. But after the event generation, the dynamic execution semantics still can activate transitions inside v. Therefore, we are not allowed to conclude that always $v \rightarrow_{hb} r$. However, from Definition 8.6, $pred_v$ always happens before v. Since r only makes a change to be fired, after v has executed at least once, $pred_v \overset{C}{\rightarrow}_{so} r$, for all possible parallel executions.

8.3.4 Transitivity

Now we define the relation $\rightarrow_{so}= (\overset{S}{\rightarrow}_{so} \cup \overset{C}{\rightarrow}_{so})^+$ as our happens before relation on statecharts. The transitivity of this order relation has an interesting property (Property 8.1), which is trivial due to the definitions of predecessors and successors.

Property 8.1. *Let $A, B \in F, x \in (\sigma_A \cup \delta_A), y \in (\sigma_B \cup \delta_B)$, and there are $D \in F, s \in \sigma_D$ such that $A, B \in \rho(s)$. The relation $\rightarrow_{so}$ on states and transitions of concurrent automata is a binary relation with the following property: if $x \rightarrow_{so} y$ then (1) $pred_x \rightarrow_{so} y$, (2) $x \rightarrow_{so} succ_y$, and (3) $pred_x \rightarrow_{so} succ_y$.*

Transitivity Illustrations

As mentioned in Section 8.1.1, broadcast events are events received by all orthogonal regions. Since all regions of a statechart are in the same object, they all receive the same events. A special type of broadcast events are *propagated events* which are events that are sent as the result of a transition taken in one orthogonal region or object. Propagated events are also (internally) broadcasted. Propagated events are extremely useful to define an ordering between the elements of three or more orthogonal regions. To do this, it is sufficient to combine propagated events with the causality of synchronization dependencies and the sequential causality inside each concurrent substate.

A first example is visualized in Figure 8.6. As you can clearly see, transitivity of the order relation is used to find the happens before relation between transition `bt2` and `ct3`. Since ordinary transitivity is used here, we will not consider such a situation in greater detail.

As another example, consider Figure 8.7. Transition `bt2` happens before transition `at0`, as derived in the previous section. The sequential flow of control in region `A` generates a happens-before relation between transition `at0` and `at5`, as visualized on the figure. It is also obvious that transition `at5` executes before transition `ct3`, due to the synchronization point between them. We are allowed to conclude that `bt2` always happens before `ct3`, if they execute in a program run. Properties 8.2-8.3 formalize this particular kind of transivity.

Property 8.2. *The relation $\overset{C}{\rightarrow}_{so}$ on states and transitions of concurrent automata has the following property: if $A, B, C \in F, r \in \delta_A, t \in \delta_B, w \in \delta_C$, and there are $D \in F$ and $s \in \sigma_D$ such that $A, B, C \in \rho(s)$ and if $r \rightarrow_{sd} t$ (Definition 7.12) and $(\exists x \in succ_r \mid w \rightarrow_{sd} x)$ then $t \overset{C}{\rightarrow}_{so} w$, for all possible parallel executions of D.*

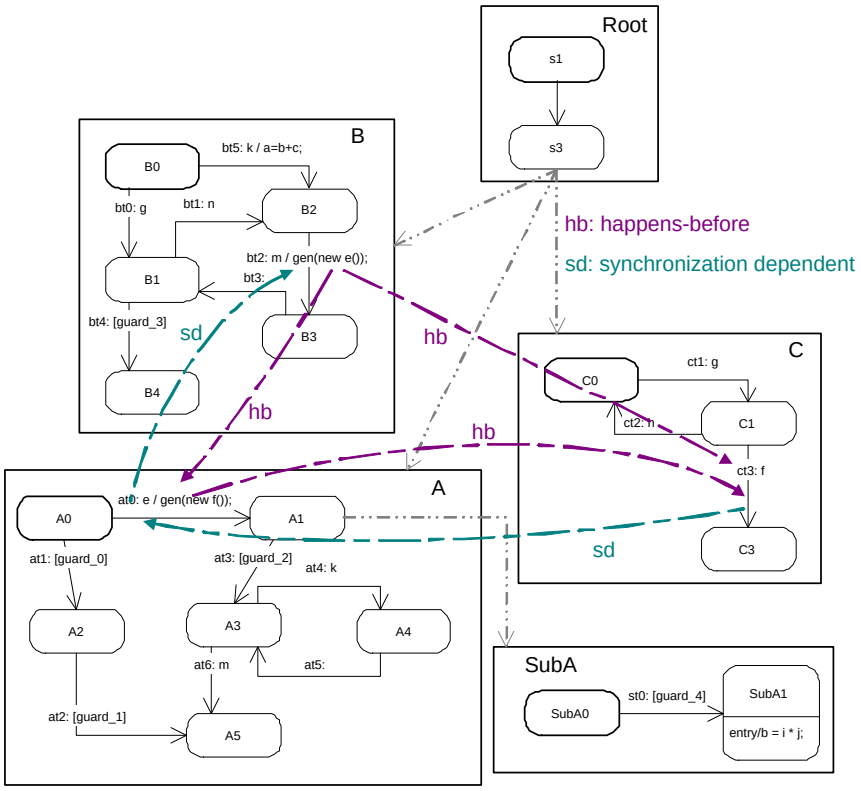


Figure 8.6: First Example of Transitivity

If $t \rightarrow_{hb} r$ then we know, from the dynamic execution semantics, that only after t is fired, transition r may be fired. From Definition 8.6, r always happens before each successor element e. If x generates an event that triggers w, then obviously t happens before w, i.e. only after t is fired, w may fire. If r executes and the machine follows a path to x, w makes a change to execute. We safely conclude that $t \rightarrow_{hb} w$, for all possible parallel executions of the concurrent state.

Property 8.3. *The relation* $\overset{C}{\rightarrow}_{so}$ *on states and transitions of concurrent automata has the following property: if* $A, B, C \in F, r \in \delta_A, v \in \sigma_B, w \in \delta_C$, *and there are* $D \in F$ *and* $s \in \sigma_D$ *such that* $A, B, C \in \rho(s)$ *and if* $r \rightarrow_{sd} v$ *(Definition 7.12) and* $(\exists x \in succ_r \mid w \rightarrow_{sd} x)$ *then* $pred_v \overset{C}{\rightarrow}_{so} w$, *for all possible parallel executions of* D.

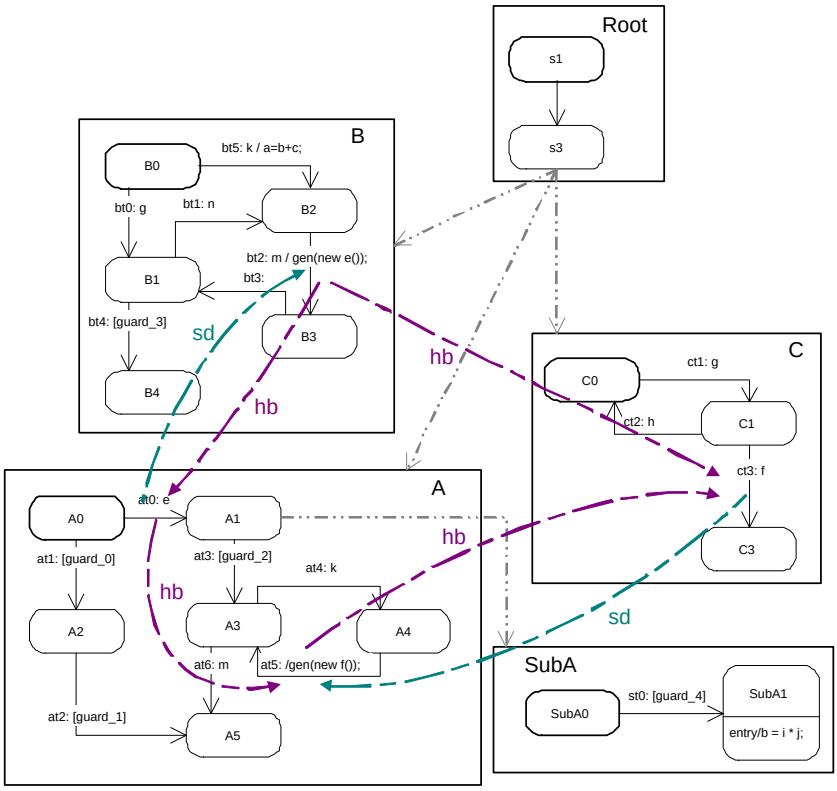


Figure 8.7: Second Example of Transitivity

8.4 Tuning Interference Dependences

The happens-before relation $\rightarrow_{so}$ gives us everything we need to refine several interference dependencies: if a happens before b, then a never can be dependent on b. Thus, an interference dependence exists, if there does not exist a happens-before relation ($\in \rightarrow_{so}$) that states the opposite. Let us give an example using Figure 8.1. Using Definition 8.9 transition `bt5` never can be parallel data dependent on state `A1`, since $(bt5, A1) \in \rightarrow_{so}$.

Definition 8.9 (Restricted Parallel Data Dependence). *If $A, B \in F$ and $u \in \sigma_A, r \in \delta_A, v \in \sigma_B, t \in \delta_B$, and there are $C \in F$ and $s \in \sigma_C$ such that $A, B \in \rho(s)$, then $u \rightarrow_{pdd} v$ (or $u \rightarrow_{pdd} t$, or $r \rightarrow_{pdd} v$, or $r \rightarrow_{pdd} t$) iff $(u.UV \cap v.DV \neq \emptyset) \wedge ((u, v) \notin \rightarrow_{so})$ (or $(u.UV \cap t.DV \neq \emptyset) \wedge ((u, t) \notin \rightarrow_{so})$, or $(r.UV \cap v.DV \neq \emptyset) \wedge ((r, v) \notin \rightarrow_{so})$, or $(r.UV \cap t.DV \neq \emptyset) \wedge ((r, t) \notin \rightarrow_{so})$*

respectively).

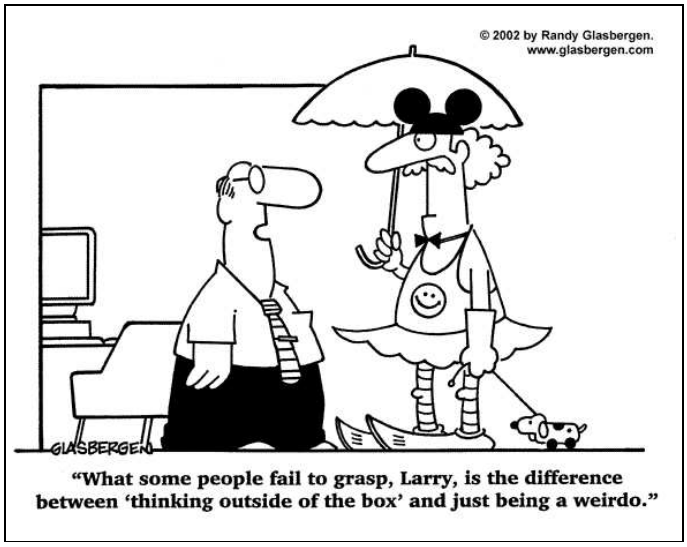
Definition 8.10 (Restricted Parallel Transition Control Dependence). *If* $A, B \in F$ *and* $r \in \delta_A, v \in \sigma_B, t \in \delta_B$, *and there are* $C \in F$ *and* $s \in \sigma_C$ *such that* $A, B \in \rho(s)$, *then* $r \rightarrow_{tcd} v$ (*or* $r \rightarrow_{tcd} t$) *iff* $(r.CV \cap v.DV \neq \emptyset) \wedge ((r, v) \notin \rightarrow_{so})$ (*or* $(r.CV \cap t.DV \neq \emptyset) \wedge ((r, t) \notin \rightarrow_{so})$ *respectively*).

The only thing that has to be done to yield an algorithm that is efficient yet effective for reducing the number of interference dependencies used in slicing statecharts with concurrent states, is letting the WDQ-algorithm use these new definitions of interference dependencies instead of the imprecise ones (Definitions 7.11 and 7.13). This also can yield a more precise slice which positively influences the verification procedure. For the statechart shown in Figure 8.1, a more precise slice will definitely be retrieved.

8.5 Conclusion

All approaches known to me to slice hierarchical automata do not consider the broadcasting mechanism that is used to let regions communicate with each other. By defining interference dependeces in terms of the broadcasting mechanism, far more precise slices are returned by the algorithm. This not only reduces the time to verify a property using the model checking technique, but also reduces the underlying datastructure (BDD) used by the model checker. This gives software developers the possibility to model check quite complex software designs. Pieces of the model that are not of interest are neglected during verification. Chapter 11 illustrates the strength of broadcasting mechanism during slicing.

Part III



Illustrations

CHAPTER 9

Model Construction in Practice

No one can whistle a symphony.
It takes an orchestra to play it.
Halford E. Luccock.

A thorough system specification is not sufficient to guarantee that the system will adequately perform its tasks during its entire life cycle. A high quality system can only be achieved when the detection of failures is started in the early design phases. To remove such failures, current practice in UML design evaluation consists of manual tedious inspections which are most of the time incomplete. Therefore, verification is highly required. Verification of design models can reveal flaws in the design level before they are implemented, and thus reduce the costs and time to market.

This manuscript has specifically addressed the process and the procedures used for an effective design verification. The technique (which is automated) allows software developers to use UML case tools for the specification of their systems, and performs the required verifications on these specifications.

The aim of this chapter is to demonstrate how part of our tool, developed in Chapters 3-4, offers effective support for verifying statechart specifications. It will be shown that the verification tool can be repeatedly integrated (and used) in an iterative and incremental design process. It will also be shown how to develop a verified design for a variant of the production cell model [76]. Some functionalities of the model will be described together with how these functionalities can be modeled (designed) using some UML diagrams. Some requirements that are to be fulfilled by the control software are described and are automatically verified. The usefulness of counterexamples to capture and to correct design failures will be demonstrated as well.

9.1 The Production Cell Model

The original production cell is in a metal processing plant in Karlsruhe, Germany. The production cell, involving a robotic system, is a realistic industry-oriented problem, where safety requirements play a significant role [76]. Forschungszentrum Informatik, Karlsruhe, used the production cell as the basis for a study of formal methods for critical software systems. It is a benchmark for evaluating methodologies for designing embedded systems. The informal description of the model is as follows:

> *... the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a travelling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt [76].*

The production cell (see Figure 9.1) has several machines that must be coordinated in order to forge metal blanks. The metal plates are taken from the *feed belt* to the *rotary table*. The *robot* takes the blanks from the table and feeds them to the press. The latter machine performs some time consuming treatment of the plate. When this job is done, the robot moves the treated blanks to the *deposit belt*.

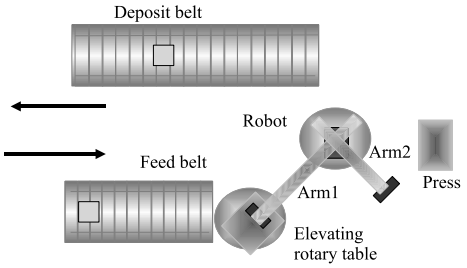


Figure 9.1: The Production Cell

We start the UML design in an object-oriented spirit. This means that we have to define the basic objects that compose the system. Obviously, the machines are working concurrently and each machine follows its own clock. Therefore, each machine will correspond to an active object. Each component internally represents a sequential process and executes its rules as soon as they become enabled. In Figure 9.2, the topview of the production cell is described.

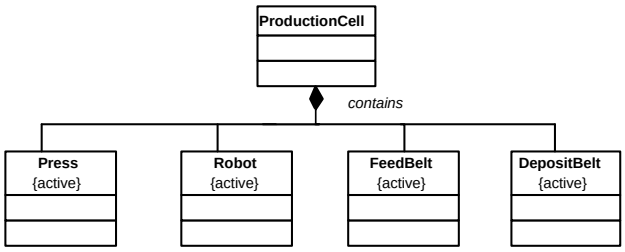


Figure 9.2: Top View Class Diagram

9.2 Modeling the Press

The press goes through the cycle of loading, forging and unloading. Initially, the press is in the middle position until the robot places a blank into the press. Then it moves upwards to forge the blank and then it moves backwards to the lower position to unload the blank. The robot gets the blank and the cycle starts again.

> *... The task for the press is to forge metal blanks. The press consists of two horizontal plates, with the lower plate being movable along a vertical axis. The press operates by pressing the lower plate against the upper plate. Because the robot arms are placed on different horizontal planes, the press has three positions. In the lower position, the press is unloaded by arm 2, while in the middle position it is loaded by arm 1. The operation of the press is coordinated with the robot arms as follows: 1. Open the press in its lower position and wait until arm 2 has retrieved the metal plate and left the press, 2. Move the lower plate to the middle position and wait until arm 1 has loaded and left the press, 3. Close the press, i.e. forge the metal plate. This processing sequence is carried out cyclically [76].*

A common way to model the press is in two parts: a controller part and a hardware part. These elements are represented by two classes: `Press_hw` and `Press_ct`. Figure 9.3 describes the relationship between these objects.

The class diagram shows the structure of the press, but it does not describe the behavior. Therefore, we need UML statechart diagrams. The behavior of the press is modeled through the behavior of its hardware part (see Figure 9.4) [78] and through the behavior of its controller part (see Figure 9.5) [78]. As can be seen, initially the press is in the middle position and the controller is in the `Loaded` state. This simply means that the press is ready to be loaded, as indicated in the informal description of the press.

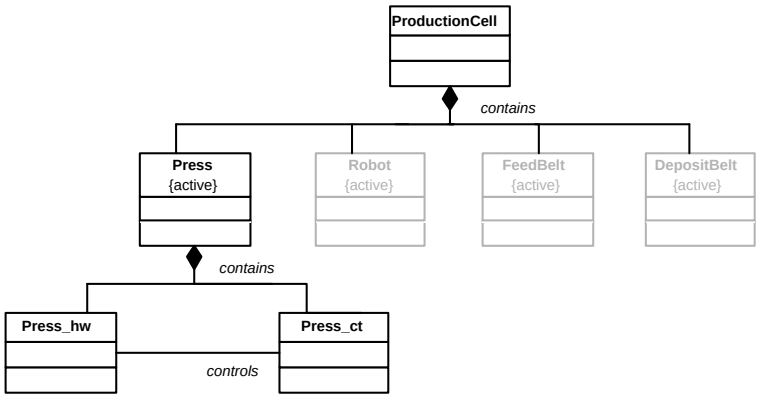


Figure 9.3: Class Diagram of the Press

At the moment the robot places a blank in the press, it sends the `forge` event to the controller. The controller reacts upon this event, moves to the `Pressing` state and instructs the hardware to move upwards.

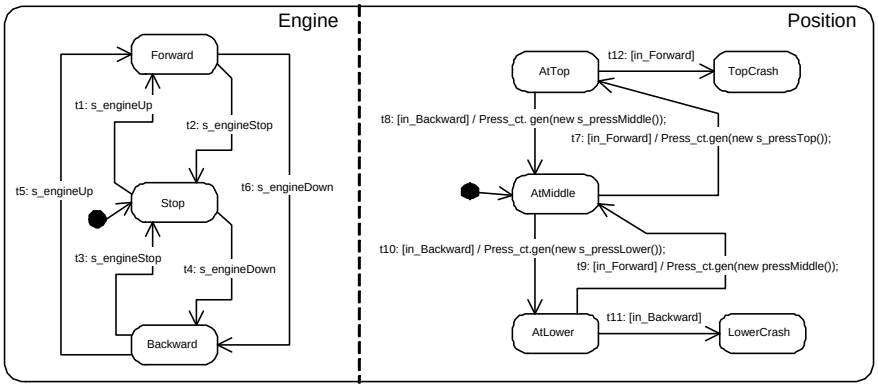


Figure 9.4: Hardware of the Press

It is important to note that the model is an *open model* with respect to the events, because we have not yet modeled the robot[1]. As an example, the event `inter_forge` has to be sent out by the robot at the moment he places a new blank into the press.

Our verification tool automatically converts open models to closed ones by activating an event generator at the appropriate moments (see Section 3.5.3

[1]We do not yet consider the entry actions attached to some states of the controller.

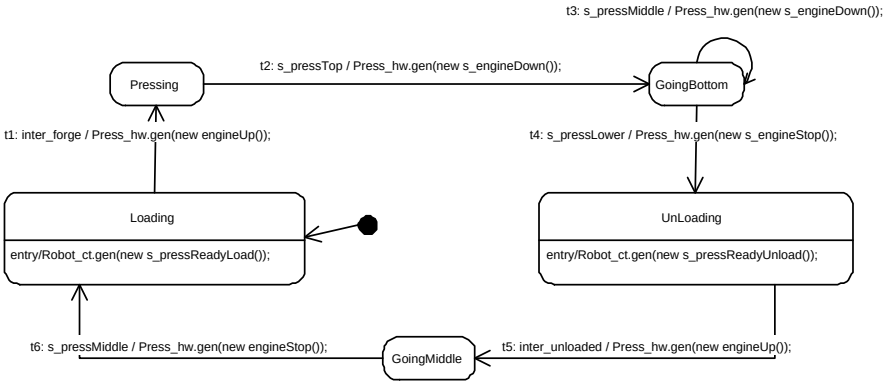


Figure 9.5: Controller of the Press

and Section 3.7.6).

9.2.1 Safety Requirement

Although the other components of the production cell are not yet modeled, the verification tool can already be used to verify some requirements that the system should meet. One such a requirement is a *safety requirement* that indicates the unreachability of a state where some property holds. In general, the requirement can be formulated in CTL as follows:

$$AG(\neg\phi) \equiv \neg EF\phi$$

A safety requirement examines every computation path to verify that ϕ never is true. As an example: the press is not moved upwards if it is in its top position [19, 76]. The requirement can be formulated in terms of a CTL formula as follows:

$$AG(\neg in\_TopCrash)$$

Once the tool has performed the necessary transformations to build a CaSMV Kripke model, it activates the model checker to automatically decide on the formula. By running the verification, a counterexample is returned; it is possible to reach the `TopCrash` state.

Table 9.1 shows the execution trace (which in fact is visualized by the tool) that leads to the violation of the constraint. To save space, only the important elements that change when the thread of the active object `Press` runs, are shown.

thrPress.	1	2	3	4	5	6	|: 7 :|
ct_st_Root	Loading	Pressing	Pressing	Pressing	Pressing	GoingBottom	GoingBottom
hw_st_Engine	Stop	Stop	Forward	Forward	Forward	Forward	Backward
hw_st_Position	AtMiddle	AtMiddle	AtMiddle	AtTop	TopCrash	TopCrash	TopCrash
event_queue[0]	inter_ct_forge	s_hw_engineUp	NotDefined	s_ct_pressTop	s_ct_pressTop	s_hw_engineDown	NotDefined
event_queue[1]	NotDefined	NotDefined	NotDefined	NotDefined	NotDefined	NotDefined	NotDefined
event_tail	1	1	0	1	1	1	0
event_overflow	0	0	0	0	0	0	0
ct_progress_auto	0	0	0	0	0	0	0
ct_progress_trigger	1	0	0	1	1	0	0
hw_progress_auto	0	0	1	1	0	0	0
hw_progress_trigger	0	1	0	0	0	1	0
object_progress	NotDefined	NotDefined	hw	hw	NotDefined	NotDefined	NotDefined
running	1	1	1	1	1	1	1

Table 9.1: Counter Example Trace

Interpretation of the counterexample At the beginning, the controller of the press is in the `Loaded` state and immediately reacts upon the event `inter_forge`; its automaton transitions to the `Pressing` state and sends the event `engineUp` to the hardware of the press. Once the hardware has received the message, its left region `Engine` first moves to the state `Forward` by taking transition `t2`. This movement causes triggerless transitions to take in the region `Position`. The latter region reaches the state `Forward` while executing transition `t7`. At the same time, it sends an event back to the controller. This event is not yet handled due to the fact that the hardware of the press is still instable (RTC-step semantics); it continues to move forward and thus reaches the state `TopCrash`. As can be seen, this state is a trap state, so everything is stuck now. The controller in its turn, handles the event `s_pressTop`, moves to the state `GoingBottom`, and finally generates the event `s_engineDown`. But since the hardware cannot react upon this event, the event is discarded and the controller remains in the state `GoingBottom` forever.

9.2.2 Invariant Related Requirement

An invariant related requirement refers to the general validity of some property ϕ. In CTL, such a requirement is formalized as follows:

$$AG\phi$$

Now, for every computation path, ϕ must always be true. As an example for the press: every unforged blank introduced into the system will eventually leave it forged [19, 76] which is defined as:

$$AG(in\_Loading \Rightarrow AF(in\_UnLoaded))$$

Since we have to deal with an open model, the requirement is formulated under the assumption that a new blank is placed into the press at the moment the controller is in the `Loaded` state. To state it differently, every time the controller reaches the `Loaded` state, a new blank is immediately inserted by immediately generating the event `inter_forge`. This is the task of the event generator. Consequently, such a way of working guarantees progress of the system, since for verification purposes, it is useless to wait a certain amount of time before a new blank is placed into the press. The same is true for the event `inter_unloaded`.Of course, since the safety requirement is not fulfilled, neither is this requirement satisfied. The same counterexample is returned.

9.2.3 Changing the Design

The problem with the design that we have so far is that the controller reacts too slowly. Thus, the design has to be changed in such a way that the reaction of the control software is fast enough to fulfil the appropriate timing requirements. To guarantee that the timing requirements are being met, we have to dramatically change the way the hardware is modeled. A more efficient way of modeling is shown in Figure 9.6 (adapted from [78]). Clearly, the regions of Figure 9.4 are joined together. This time the safety and invariant requirement are met by the model, as wanted. Note that the transitions to the `TopCrash` and `LowerCrash` state will never be taken; they are in a way redundant.

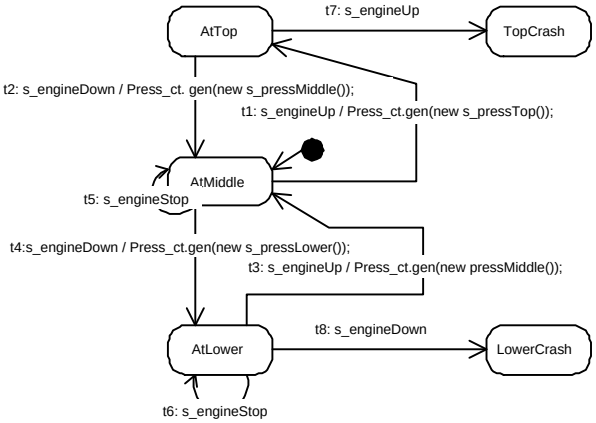


Figure 9.6: Revised Hardware of the Press

9.3 Modeling the Robot

The robot component, as central unit, is the most complex machine of the production cell. The robot has two arms which sometimes move together. Just like the press, the behavior of the robot is cyclic:

> *... The robot's task consists in: taking metal blanks from the elevating rotary table to the press; transporting forged plates from the press to the deposit belt [76].*

The robot first waits for the rotary table to be in the correct position, and for a plate to be present on it. Once these conditions are fulfilled, the robot

unloads the table using the first arm. Its second arm moves towards the press to take a forged blank from it. The robot loads the deposit belt by moving its second arm from the press towards the belt. Of course, the robot possibly has to wait for the press to be free. The robot feeds the press with a new blank using its first arm. The cycle now restarts.

As before, the robot is modeled by two parts: a controller part and a hardware part, as shown in Figure 9.7. If we leave out all the details of the arm-movements, we get a cyclic abstract behavior of the robot (adapted from [19, 78]), as shown in Figure 9.8. Initially, it is assumed that the robot is in the `UnLoadTable` state.

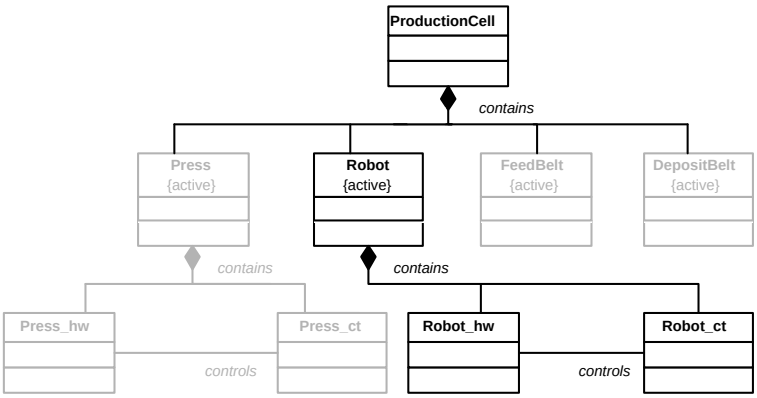


Figure 9.7: Class Diagram of the Robot

9.3.1 Deadlock Free Requirement

As can directly be seen on Figure 9.8, there is only communication (through events) with the press, since the belts are not yet modeled. Is the system modeled so far deadlock free? Generally speaking, a concurrent program is in a *deadlock* situation when no terminal state is reached, and no part of the program is able to proceed. A system is *deadlock-free* when no execution leads to a deadlock. Deadlock freedom can be expressed by the following formula:

$$AG(\neg deadlock)$$

The production cell deadlocks when no statechart is able to trigger a transition. This is specified as follows:

$$AG(AF(\neg(not\_progress\_auto \wedge not\_progress\_trigger)))$$

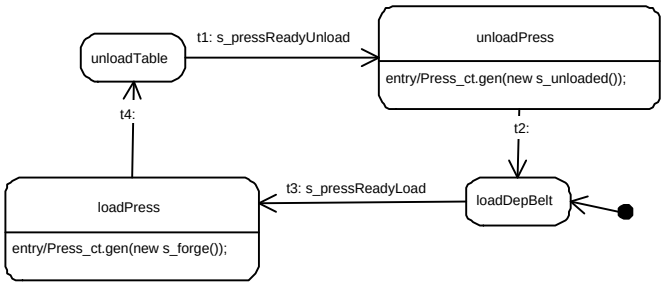


Figure 9.8: Abstract Behavior of the Robot's Controller

with

$$
\begin{array}{lcll}
not\_progress\_auto & := & & (thrPress.hw\_progress\_auto = 0) \\
 & & \wedge & (thrPress.ct\_progress\_auto = 0) \\
 & & \wedge & (thrRobot.ct\_progress\_auto = 0) \\
not\_progress\_trigger & := & & (thrPress.hw\_progress\_trigger = 0) \\
 & & \wedge & (thrPress.ct\_progress\_trigger = 0) \\
 & & \wedge & (thrRobot.ct\_progress\_trigger = 0)
\end{array}
$$

The system does not deadlock when either the press makes progress or the robot. CaSMV verifies that the system, modeled so far, satisfies the deadlock free requirement but only when it is assumed that the initial state of the controller is the `loadDepBelt` state. Of course, the safety and invariant requirements are still being met by the system.

9.3.2 Refining the Design

At this point, we can refine the robot's behavioral model by refining each abstract state. Such a refinement takes both the movements of the arms into account and the position of the corresponding components. Further details can be found in [19, 78].

9.4 A Word on the CaSMV Kripke Model

As must already be clarified by now, software developers use our verification tool to automatically decide on the requirements. One of the responsibilities of the tool is to create a CaSMV Kripke model, which is the input of the model checker CaSMV. Since we have several active objects, the CaSMV model looks like given in the code listings below.

```
#define SIZE 3
module main() {

  thrPress_event_queue: array 0..(SIZE-1) of
              {s_hw_engineUp, s_hw_engineDown,
               s_hw_engineStop, s_ct_forge,
               s_ct_unloaded, s_ct_pressMiddle,
               s_ct_pressTop, s_ct_pressLower, NotDefined};
  thrPress_event_tail: 0..(SIZE-1);
  thrPress_event_overflow: boolean;

  init(thrPress_event_queue[0]):= {NotDefined};
  init(thrPress_event_queue[1]):= {NotDefined};
  init(thrPress_event_queue[2]):= {NotDefined};
  init(thrPress_event_tail):= 0;
  init(thrPress_event_overflow):= 0;

  thrRobot_event_queue: array 0..(SIZE-1) of
       {s_ct_pressReadyLoad, s_ct_pressReadyUnLoad, NotDefined};
  thrRobot_event_tail: 0..(SIZE-1);
  thrRobot_event_overflow: boolean;

  init(thrRobot_event_queue[0]):= {s_ct_pressReadyLoad};
  init(thrRobot_event_queue[1]):= {NotDefined};
  init(thrRobot_event_queue[2]):= {NotDefined};
  init(thrRobot_event_tail):= 1;
  init(thrRobot_event_overflow):= 0;

  thrPress:process Press(thrPress_event_queue, thrPress_event_tail,
                thrPress_event_overflow, thrRobot_event_queue,
                thrRobot_event_tail, thrRobot_event_overflow);
  thrRobot:process Robot(thrPress_event_queue, thrPress_event_tail,
                thrPress_event_overflow, thrRobot_event_queue,
                thrRobot_event_tail, thrRobot_event_overflow);
  thrFeedBelt: process FeedBelt();
  thrDepositBelt: process DepositBelt();

  safety         : SPEC AG (~thrPress.hw_st_Root=TopCrash);
  invariant      : SPEC AG (thrPress.ct_st_Root=Loading ->
                                AF(thrPress.ct_st_Root=UnLoading));
  deadlockFree: SPEC AG ( AF (~(thrPress.hw_progress_auto=0     &
                                thrPress.ct_progress_auto=0     &
                                thrPress.hw_progress_trigger=0  &
```

```
                                thrPress.ct_progress_trigger=0  &
                                thrRobot.ct_progress_auto=0     &
                                thrRobot.ct_progress_trigger=0)));
}
```

Above we see that the CaSMV model consists of four modules (since there are four active objects) and that the module of the press contains the state machine of both the hardware and the controller. The same will be true for the other modules once their corresponding design is completed. Of course, since the concurrent objects communicate (asynchronously) with each other, their event queues are globally declared and globally initialized (i.e. in the `main module`). That way, changes to the queue done in one process are visible in the other processes. The initialization of the robot event queue looks special but it isn't: Everytime the press enters the `Loading` state, the event `s_ct_pressReadyLoad` is generated and sent to the robot. Since the `Loading` state is an initial state, this is also done when the system starts up. Remark that the events `inter_forge` and inter_unloaded became signal events now; the system itself is able to generate them, making the event generator redundant.

```
module Press(...) {

  thrPress_event_queue: {...};
  thrPress_event_tail: 0..(SIZE-1); thrPress_event_overflow: boolean;
  thrRobot_event_queue: {...};
  thrRobot_event_tail: 0..(SIZE-1); thrRobot_event_overflow: boolean;

  hw_st_Root: {AtTop, AtMiddle, AtLower,
               LowerCrash, TopCrash, NotActive};
  ct_st_Root: {Pressing, Loading, GoingMiddle,
               UnLoading, GoingBottom, StateMachineError};

  in_hw_AtTop, in_hw_AtMiddle, in_hw_AtLower, ...: boolean;
  in_hw_AtTop     := hw_st_Root = AtTop;
  in_hw_AtMiddle  := hw_st_Root = AtMiddle;
  in_hw_AtLower   := hw_st_Root = AtLower;
  ...;

  in_ct_Pressing, in_ct_Loading, in_ct_GoingMiddle, ...: boolean;
  in_ct_Pressing   := ct_st_Root = Pressing;
  in_ct_Loading    := ct_st_Root = Loading;
  in_ct_GoingMiddle:= ct_st_Root = GoingMiddle;
  ...;
```

```
error: boolean;
error:= thrPress_event_overflow;

init(hw_st_Root)    := AtMiddle;

init(ct_st_Root):= Loading;

hw_t1, hw_t2, hw_t3, hw_t4, hw_t5, hw_t6, hw_t7, hw_t8: boolean;
hw_t1:=in_hw_AtMiddle & thrPress_event_queue[0]=s_hw_engineUp;
hw_t2:=in_hw_AtTop & thrPress_event_queue[0]=s_hw_engineDown;
...;

ct_t1, ct_t2, ct_t3, ct_t4, ct_t5, ct_t6: boolean;
ct_t1:=in_ct_Loading & thrPress_event_queue[0] = s_ct_forge;
ct_t2:=in_ct_Pressing & thrPress_event_queue[0] = s_ct_pressTop;
...;

hw_progress_auto, ct_progress_auto: boolean;
hw_progress_auto:= 0; ct_progress_auto:= 0;

hw_progress_trigger, ct_progress_trigger: boolean;
hw_progress_trigger:= hw_t1 | ... | hw_t8;
ct_progress_trigger:= ct_t1 | ... | ct_t6;

last: {hw, ct, NotDefined};
init(last):= ct;

object_progress: {hw, ct, NotDefined};
object_progress:= case {
    last = hw & hw_progress_auto: hw;
    last = ct & ct_progress_auto: ct;
    ~(hw_progress_auto | ct_progress_auto): NotDefined;
    default: {hw_progress_auto ? hw, ct_progress_auto ? ct};
};
```

Special to note is the initialization of the variable `last` which refers to the object that made progress in the previous time step. When the system starts up, all the statecharts enter their initial state. So in fact, we cannot say if it is either the hardware that has entered its initial state before the controller or the other way around. We claim it to be the controller since the initial state of the controller immediately communicates with the robot by sending an event.

```
case {

  object_progress=hw & ~error: {
     next(ct_st_Root):= ct_st_Root;
     next(hw_st_Root):= hw_st_Root;
     next(thrPress_event_queue):= thrPress_event_queue;
     next(thrPress_event_tail):= thrPress_event_tail;
     next(thrPress_event_overflow):= thrPress_event_overflow;
     next(thrRobot_event_queue):= thrRobot_event_queue;
     next(thrRobot_event_tail):= thrRobot_event_tail;
     next(thrRobot_event_overflow):= thrRobot_event_overflow;
     next(last):=hw;
  };

  object_progress=ct & ~error: {
     next(hw_st_Root):= hw_st_Root;
     next(ct_st_Root):= ct_st_Root;
     next(thrPress_event_queue):= thrPress_event_queue;
     next(thrPress_event_tail):= thrPress_event_tail;
     next(thrPress_event_overflow):= thrPress_event_overflow;
     next(thrRobot_event_queue):= thrRobot_event_queue;
     next(thrRobot_event_tail):= thrRobot_event_tail;
     next(thrRobot_event_overflow):= thrRobot_event_overflow;
     next(last):= ct;
  };
```

Since our tool works automatically, it generates the above two code chunks. In fact, they can be omitted because they never will be taken (Figure 9.5 and Figure 9.6). It is not that difficult to integrate such an optimization in the tool. Below, it is illustrated how the event queue of the thread is updated.

```
  hw_progress_trigger & ~error: {
     next(thrRobot_event_queue)   := thrRobot_event_queue;
     next(thrRobot_event_tail)    := thrRobot_event_tail;
     next(thrRobot_event_overflow):= thrRobot_event_overflow;
     next(ct_st_Root):= ct_st_Root;
     next(hw_st_Root):= case {
                          hw_t1: AtTop;
                          hw_t2|hw_t3|hw_t5: AtMiddle;
                          hw_t4|hw_t6: AtLower;
                          hw_t7: TopCrash;
                          hw_t8: LowerCrash;
                        };
```

```
for(i=0; i<SIZE; i=i+1) {
  if (((i+1)<thrPress_event_tail) & (i+1)<SIZE)
    next(thrPress_event_queue[i]):=thrPress_event_queue[i+1];
  else {
   if(~(thrPress_event_tail=0)) {
     case {
       hw_t1 & thrPress_event_tail = (i+1):
           next(thrPress_event_queue[i]):=s_ct_pressTop;
       (hw_t2|hw_t3) & thrPress_event_tail = (i+1):
           next(thrPress_event_queue[i]):=s_ct_pressMiddle;
       hw_t4 & thrPress_event_tail = (i+1):
           next(thrPress_event_queue[i]):=s_ct_pressLower;
       default: next(thrPress_event_queue[i]):= NotDefined;
     };
   } else {
         case {
           hw_t1 & thrPress_event_tail = (i):
             next(thrPress_event_queue[i]):=s_ct_pressTop;
           (hw_t2|hw_t3) & thrPress_event_tail = (i):
             next(thrPress_event_queue[i]):=s_ct_pressMiddle;
           hw_t4 & thrPress_event_tail = (i):
             next(thrPress_event_queue[i]):=s_ct_pressLower;
           default: next(thrPress_event_queue[i]):=
              thrPress_event_queue[i];
         };
   };
  };
};

if(~(thrPress_event_tail=0)) {
 case {
   hw_t1|hw_t2|hw_t3|hw_t4:
       if (((thrPress_event_tail-1)+1)>=SIZE) {
          next(thrPress_event_tail):=thrPress_event_tail;
          next(thrPress_event_overflow):= 1;
       } else {
      next(thrPress_event_tail):=(thrPress_event_tail-1)+1;
      next(thrPress_event_overflow):=0;
            };

   hw_t5|hw_t6|hw_t7|hw_t8: {
     next(thrPress_event_tail):=  thrPress_event_tail-1;
```

```
            next(thrPress_event_overflow):= 0;
                  };
           default: {
           next(thrPress_event_tail):=  thrPress_event_tail;
           next(thrPress_event_overflow):= thrPress_event_overflow;
                    };
      };
      } else {
           case {
             hw_t1|hw_t2|hw_t3|hw_t4:
                 if (((thrPress_event_tail)+1)>=SIZE) {
              next(thrPress_event_tail):= thrPress_event_tail;
              next(thrPress_event_overflow):= 1;
             } else {
              next(thrPress_event_tail):= (thrPress_event_tail)+1;
              next(thrPress_event_overflow):= 0;
                 };

           default: {
           next(thrPress_event_tail):=  thrPress_event_tail;
           next(thrPress_event_overflow):= thrPress_event_overflow;
                    };
            };
      };

      next(last):= hw;
    };
```

The controller of the press performs communication with the robot. Therefore, at appropriate moments, the controller places those events into the queue of the robot. Only then, the robot is able to react upon them. The code listing below clearly demonstrates this. However note the difference in the way the queue of the robot and the queue of the press is updated. In the former one, it is sufficient to just add a new event following the FIFO semantics. Evidently, it is forbidden to shift out the first event of the queue of the robot; we are working in the thread of the press. The latter removes the head of the queue (dispatched and processed), and adds new events at the end of the queue.

```
ct_progress_trigger & ~error: {
  next(hw_st_Root):= hw_st_Root;
  next(ct_st_Root):= case {
                       ct_t1: Pressing;
```

```
                    ct_t2 | ct_t3: GoingBottom;
                    ct_t4: UnLoading;
                    ct_t5: GoingMiddle;
                    ct_t6: Loading;
                   };

for(i=0; i<SIZE; i=i+1) {
  case {
    ct_t4 & thrRobot_event_tail = i:
     next(thrRobot_event_queue[i]):= s_ct_pressReadyUnLoad;
    ct_t6 & thrRobot_event_tail = i:
     next(thrRobot_event_queue[i]):= s_ct_pressReadyLoad;
    default: next(thrRobot_event_queue[i]):=
     thrRobot_event_queue[i];
  };
};

case {
  ct_t4 | ct_t6: if (((thrRobot_event_tail)+1)>=SIZE) {
    next(thrRobot_event_tail)    := thrRobot_event_tail;
    next(thrRobot_event_overflow):= 1;
                } else {
     next(thrRobot_event_tail)    := (thrRobot_event_tail)+1;
     next(thrRobot_event_overflow):= 0;
                   };
  default: {
    next(thrRobot_event_tail):=  thrRobot_event_tail;
    next(thrRobot_event_overflow):= thrRobot_event_overflow;
           };
};

for(i=0; i<SIZE; i=i+1) {
  if (((i+1)<thrPress_event_tail) & (i+1)<SIZE)
  next(thrPress_event_queue[i]):= thrPress_event_queue[i+1];
  else {
    if(~(thrPress_event_tail=0)) {
      case {
        (ct_t1|ct_t5) & thrPress_event_tail = (i+1):
            next(thrPress_event_queue[i]):= s_hw_engineUp;
        (ct_t2|ct_t3) & thrPress_event_tail = (i+1):
            next(thrPress_event_queue[i]):= s_hw_engineDown;
        ct_t4 & thrPress_event_tail = (i+1):
            next(thrPress_event_queue[i]):= s_hw_engineStop;
```

```
          ct_t6 & thrPress_event_tail = (i+1):
              next(thrPress_event_queue[i]):= s_hw_engineStop;
          default: next(thrPress_event_queue[i]):= NotDefined;
        };
      } else {
      case {
        (ct_t1|ct_t5) & thrPress_event_tail = (i):
            next(thrPress_event_queue[i]):= s_hw_engineUp;
        (ct_t2|ct_t3) & thrPress_event_tail = (i):
            next(thrPress_event_queue[i]):= s_hw_engineDown;
        ct_t4 & thrPress_event_tail = (i):
            next(thrPress_event_queue[i]):= s_hw_engineStop;
        ct_t6 & thrPress_event_tail = (i):
            next(thrPress_event_queue[i]):= s_hw_engineStop;
        default: next(thrPress_event_queue[i]):=
            thrPress_event_queue[i];
      };
      };
   };
};

if (~(thrPress_event_tail=0)) {
 case {
   ct_t4|ct_t6: if (((thrPress_event_tail-1)+1)>=SIZE) {
        next(thrPress_event_tail):= thrPress_event_tail;
        next(thrPress_event_overflow):= 1;
                  } else {
  next(thrPress_event_tail):= (thrPress_event_tail-1)+1;
  next(thrPress_event_overflow):= 0;
                  };
  default:  {
   next(thrPress_event_tail):=  thrPress_event_tail-1+1;
   next(thrPress_event_overflow):= thrPress_event_overflow;
            };
 };
 }  else {
      case {
         ct_t1|ct_t2|ct_t3|ct_t5:
             if (((thrPress_event_tail)+1)>=SIZE) {
          next(thrPress_event_tail):= thrPress_event_tail;
          next(thrPress_event_overflow):= 1;
             } else {
    next(thrPress_event_tail):= (thrPress_event_tail)+1;
```

```
          next(thrPress_event_overflow):= 0;
                    };
          ct_t4|ct_t6: if ((thrPress_event_tail+1)>=SIZE) {
               next(thrPress_event_tail):= thrPress_event_tail;
               next(thrPress_event_overflow):= 1;
                           } else {
          next(thrPress_event_tail):= thrPress_event_tail+1;
          next(thrPress_event_overflow):= 0;
                          };
          default:  {
           next(thrPress_event_tail):=  thrPress_event_tail;
           next(thrPress_event_overflow):= thrPress_event_overflow;
                    };
          };
        };

      next(last):= ct;
    };

    error: {
      next(hw_st_Root) := StateMachineError;
      next(ct_st_Root) := StateMachineError;
      next(thrPress_event_queue):= thrPress_event_queue;
      next(thrPress_event_tail):= thrPress_event_tail;
      next(thrPress_event_overflow):= thrPress_event_overflow;
      next(thrRobot_event_queue):= thrRobot_event_queue;
      next(thrRobot_event_tail):= thrRobot_event_tail;
      next(thrRobot_event_overflow):= thrRobot_event_overflow;
      next(last):= last;
    };

    default: {
       next(hw_st_Root):= hw_st_Root;
       next(ct_st_Root):= ct_st_Root;
       next(thrPress_event_queue):= thrPress_event_queue;
       next(thrPress_event_tail):= thrPress_event_tail;
       next(thrPress_event_overflow):= thrPress_event_overflow;
       next(thrRobot_event_queue):= thrRobot_event_queue;
       next(thrRobot_event_tail):= thrRobot_event_tail;
       next(thrRobot_event_overflow):= thrRobot_event_overflow;
       next(last):= last;
    };
  };
```

```
  FAIRNESS running;
}
```

The CaSMV Kripke model for the robot looks similar. Giving further details about this model is therefore omitted.

```
module Robot(...) {

  thrPress_event_queue: array 0..(SIZE-1) of {...};
  thrPress_event_tail: 0..(SIZE-1); thrPress_event_overflow: boolean;
  thrRobot_event_queue: array 0..(SIZE-1) of {...};
  thrRobot_event_tail: 0.(.SIZE-1); thrRobot_event_overflow: boolean;

  ct_st_Root: {unloadTable, unloadPress, loadDepBelt,
                      loadPress, StateMachineError};

  in_unloadTable, in_unloadPress, ...: boolean;
  in_unloadTable := ct_st_Root=unloadTable;
  in_unloadPress := ct_st_Root=unloadPress;
  ...;

  error: boolean;
  error:= thrRobot_event_overflow;

  init(ct_st_Root):= loadDepBelt;

  ct_t1, ct_t2, ct_t3, ct_t4: boolean;
  ct_t1:= in_unloadTable &
          thrRobot_event_queue[0]=s_ct_pressReadyUnLoad;
  ct_t2:= in_unloadPress; ct_t4:= in_loadPress;
  ct_t3:= in_loadDepBelt &
          thrRobot_event_queue[0]=s_ct_pressReadyLoad;

  ct_progress_auto, ct_progress_trigger: boolean;
  ct_progress_auto   := ct_t2 | ct_t4;
  ct_progress_trigger:= ct_t1 | ct_t3;

  case {
    ct_progress_auto & ~error: {
      next(thrPress_event_queue):= thrPress_event_queue;
      next(thrPress_event_tail):= thrPress_event_tail;
      next(thrPress_event_overflow):= thrPress_event_overflow;
      next(thrRobot_event_queue) := thrRobot_event_queue;
```

```
  next(thrRobot_event_tail)  := thrRobot_event_tail;
  next(thrRobot_event_overflow):= thrRobot_event_overflow;

  next(ct_st_Root):= case {
                       ct_t2: loadDepBelt;
                       ct_t4: unloadTable;
                       default: ct_st_Root;
                     };
};

ct_progress_trigger & ~error: {
  next(ct_st_Root):= case {
                       ct_t1: unloadPress;
                       ct_t3: loadPress;
                       default: ct_st_Root;
                     };

  for(i=0; i<SIZE; i=i+1) {
    if (((i+1)<thrRobot_event_tail) & (i+1)<SIZE)
     next(thrRobot_event_queue[i]):= thrRobot_event_queue[i+1];
    else next(thrRobot_event_queue[i]):= NotDefined;
  };

  next(thrRobot_event_tail):= thrRobot_event_tail - 1;
  next(thrRobot_event_overflow):= thrRobot_event_overflow;

  for(i=0; i<SIZE; i=i+1) {
    case {
      ct_t1 & thrPress_event_tail = i:
              next(thrPress_event_queue[i]):= s_ct_unloaded;
      ct_t3 & thrPress_event_tail = i:
              next(thrPress_event_queue[i]):= s_ct_forge;
      default: next(thrPress_event_queue[i]):=
              thrPress_event_queue[i];
    };
  };

   case {
     ct_t1|ct_t3: if (((thrPress_event_tail)+1)>=SIZE){
        next(thrPress_event_tail):= thrPress_event_tail;
        next(thrPress_event_overflow):= 1;
                  } else {
     next(thrPress_event_tail):=  thrPress_event_tail+1;
```

```
          next(thrPress_event_overflow):= thrPress_event_overflow;
                      };
          default: {
           next(thrPress_event_tail):= thrPress_event_tail;
           next(thrPress_event_overflow):= thrPress_event_overflow;
                  };
      };
    };

    error: {
      next(thrPress_event_queue):= thrPress_event_queue;
      next(thrPress_event_tail):= thrPress_event_tail;
      next(thrPress_event_overflow):= thrPress_event_overflow;
      next(thrRobot_event_queue):= thrRobot_event_queue;
      next(thrRobot_event_tail):= thrRobot_event_tail;
      next(thrRobot_event_overflow):= thrRobot_event_overflow;
      next(ct_st_Root):= StateMachineError;
    };

    default: {
      next(thrPress_event_queue):= thrPress_event_queue;
      next(thrPress_event_tail):= thrPress_event_tail;
      next(thrPress_event_overflow):= thrPress_event_overflow;
      next(thrRobot_event_queue):= thrRobot_event_queue;
      next(thrRobot_event_tail):= thrRobot_event_tail;
      next(thrRobot_event_overflow):= thrRobot_event_overflow;
      next(ct_st_Root):= ct_st_Root;
    };
  };

  FAIRNESS running;
}

module FeedBelt() {

  FAIRNESS running;
}

module DepositBelt() {

  FAIRNESS running;
}
```

9.5 Conclusion

This chapter has shown that UML statechart diagrams can be used to model several parts of the production cell. The chapter has also shown that behavioral requirements are automatically verified by our tool. This means that a software developer can easily validate his design, since the entire background mathematics will be hidden from him. However, it is up to him to interpret the returned visualized counterexample and to change the design in such a way that the failure is removed from the behavioral model. It may not be a surprise that some failures are automatically detected by the tool (e.g. queue overruns, etc.) and that other requirements have to be specified by the developers (e.g. safety requirements, invariant related requirements, etc.). Consequently, we have shown the usefulness of our tool in an iterative and incremental design process.

The methodology presented here is not unknown in the literature. For example, in [78] it is explained how to use the Unified Modeling Language to model the control software of the production cell. Instead of using CaSMV, they use SPIN during the verification of their requirements. One big disadvantage of [78] is that the kind of models is strongly restricted. Using the *vUML* verification tool [77] requires UML models that consist only of concurrent objects. To state it differently, each object that comes equipped with a statechart has to be an active object. If not, the tool is not able to verify anything at all. Our work does not have such a limitation and therefore allows richer and more complete UML models. Moreover, a design consisting of both active and passive objects that communicate (a)synchronously with each other is far more realistic and needed than designs containing active objects only. We claim it to be one of the strengths of our tool.

In [18, 32] a more complete UML model for the production cell is developed. Beside statechart diagrams, use case diagrams, sequence diagrams, component diagrams, etc. are part of the model. The model is translated to SPIN as well. The main disadvantage (also present in [24]) is the same as the one of [78].

CHAPTER 10

Slicing Theory in Practice

It doesn't happen till you do.
Alan Horvath.

An important phase in the design of reactive systems is the verification of their behavioral statecharts as the application domain requires very high quality systems. A thorough verification using a model checker is a very costly procedure because of the state explosion problem. One way to overcome this problem is to combine slicing with model checking. When slicing is applied to the design to ease the model checking procedure then it must be related to the properties that are to be verified: for a given property, the slice (i.e. a smaller statechart) is the set of sequential automata that influence the verification results of the properties in some way. Naturally, the satisfaction of the properties must be the same for both models.

Chapter 7 has illustrated a slicing algorithm, more precisely the WDQ-algorithm, which removes irrelevant parts of behavioral statecharts with respect to a property to be verified. It is the purpose of this chapter to show how the algorithm works in practice using the coffee vending machine example (see Section 3.3). To do that, different temporal properties are to be considered. It will be shown that the algorithm only removes parts of the statechart when it is allowed to. As a consequence, the sliced model is not always smaller than the model of the design. Intuitively, it will become clear that this all depends on the property to be verified and, of course, on the way the statechart is designed.

10.1 Is State `CupIdle` Reachable?

Let us illustrate the slicing algorithm on the coffee vending machine (see Section 3.3) example. For clarity reasons, Figure 10.1 shows the corresponding annoted EHA.

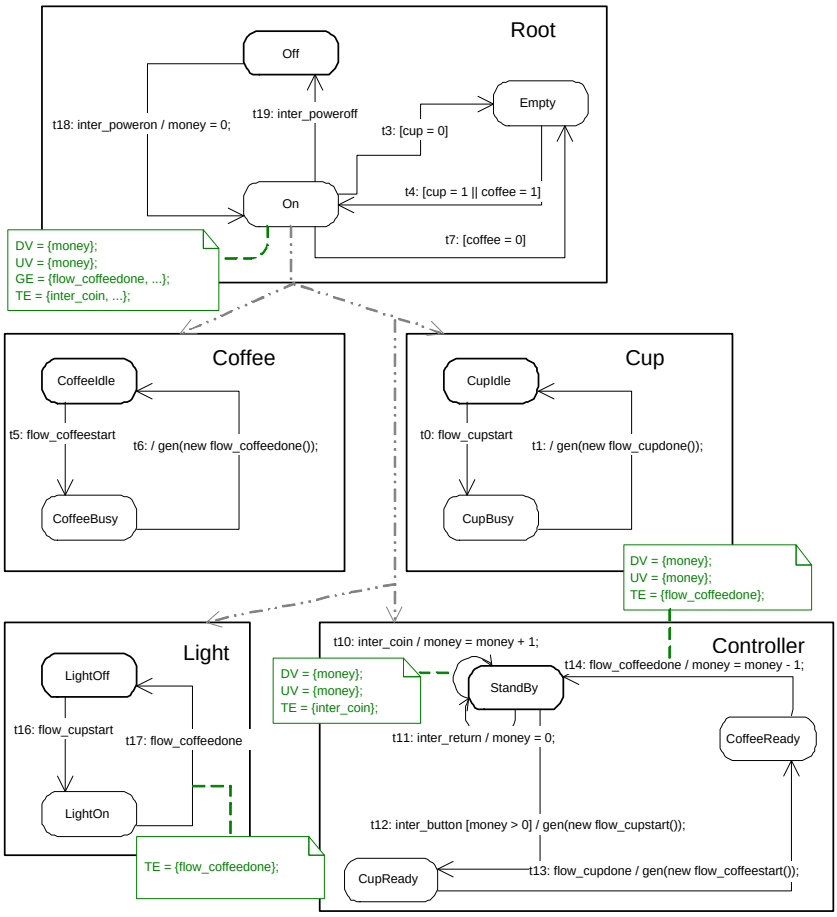


Figure 10.1: Annoted EHA for the Coffee Vending Machine

Suppose now that we want to verify a trivial property stating that at some point during execution the machine must reach the state `CupIdle`. In LTL, this will be expressed as follows:

$$F(st\_Cup = CupIdle)$$

The slicing criterion extracted from this property will be $< \{CupIdle\}, \emptyset >$. That way it is realized that the state `CupIdle` is present in the slice. Remember that every state/transition affecting the slicing criterion should be present in the slice; otherwise, the resulting statechart is not functionally equivalent to the original statechart with respect to the property in model checking UML statecharts. If such a state/transition was missing from the slice, it could result in a situation where the model checker states that a property holds on the sliced statechart even though it does not hold on the original statechart.

As can directly be seen on Figure 10.1, the sequential automaton `Light` will not be present in the slice returned by the WDQ-algorithm. The reason for this is quite obvious; the automaton `Light` only indicates progress of the vending machine; it does not control the behavior of the machine. The presence or the absence of this sequential automaton really does not influence the satisfaction of the property $F(st\_Cup = CupIdle)$.

We will now show in detail how the WDQ-algorithm computes the slice.

10.1.1 Iteration 1

At the first step, the algorithm adds only the states and the transitions of the slicing criterion to the final slice. Of course, `CupIdle` belongs to the region `Cup` and therefore all the elements of `Cup` are added to the sets ES and ET respectively. Remember that this is done to guarantee that the slice is functionally equivalent to the original model.

Iteration 1: Step 1 of the WDQ-algorithm

1. Given the criterion $< \{CupIdle\}, \emptyset >$. Then

$RS = \{CupIdle\} \quad Refine_R(CupIdle) = True$
$RT = \emptyset$

$ES = \{CupBusy\} \quad Refine_R(CupBusy) = False$
$ET = \{t0, t1\} \quad Refine_R(t0) = False$
$\qquad\qquad\quad Refine_R(t1) = False$

$RS = RS \cup ES = \{CupIdle\} \cup \{CupBusy\} = \{CupIdle, CupBusy\}$
$RT = RT \cup ET = \{t0, t1\}$
$IS = RS = \{CupIdle, CupBusy\}$
$IT = RT = \{t0, t1\}$

The algorithm now goes into the second step of the algorithm. It is important to find the elements on which the elements of IS and IT depend. The algorithm only adds the state `On` and the transition `t12` to NS and NT respectively.

Iteration 1: Step 2 of the WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{On\}$ since $CupIdle \rightarrow_{rcd} On \wedge CupIdle \in IS \wedge Refine_R(CupIdle)$
$NT = \{t12\}$ since $t0 \rightarrow_{sd} t12 \wedge t0 \in IT \wedge \neg Refine_R(t0)$

The third step of the algorithm is now being processed. It is only possible to reach state `On` along some execution path iff all the other elements of the root automaton are present in the final slice. The same can be said for transition `t12`. Summarized, the step is needed for the soundness aspect of the slicing algorithm.

Iteration 1: Step 3 of the WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \{Off, Empty, CupReady, StandBy, CoffeeReady\}$
$ET = \{t3, t7, t18, t4, t19, t14, t11, t10, t13\}$

Thereafter, the most tricky step is executed. The fourth step of the algorithm refreshes the sets IS and IT. But in this iteration nothing special happens. Note that the initial states `Off` and `StandBy` are included in the set IS. These states will be treated separately in the second step of the algorithm (next iteration). For such an initial state the refinement control dependence is extremely important; it gives us the opportunity to include states to the final slice that are higher in the hierarchy. The remaining steps 5, 6 and 7 are easy to understand.

Iteration 1: Step 4 of the WDQ-algorithm

4. Reconstruct IS and IT:

$IS = (NS \setminus RS) \cup \{Off, StandBy\} = \{On\} \cup \{Off, StandBy\}$
$IT = (NT \setminus RT) = NT \cup ET = \{t3, t7, t18, t4, t19, t14, t11, t10, t13, t12\}$

Iteration 1: Step 5, 6 and 7 of the WDQ-algorithm

5. $NS \cup ES = \{On\} \cup \{Off, Empty, CupReady, StandBy, CoffeeReady\}$
$NT \cup ET = \{t12\} \cup \{t3, t7, t18, t4, t19, t14, t11, t10, t13\}$

$Refine_R(On) = True \wedge On \in NS$
$Refine_R(Off) = False \wedge Off \notin NS$
...
$Refine_R(CoffeeReady) = False \wedge CoffeeReady \notin NS$

$Refine_R(t12) = True \wedge t12 \in NT$
$Refine_R(t3) = False \wedge t3 \notin NT$
...
$Refine_R(t13) = False \wedge t13 \notin NT$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES = \{CupIdle, CupBusy\} \cup NS \cup ES$
$RT = RT \cup NT \cup ET = \{t0, t1\} \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

10.1.2 Iteration 2

Since either IS nor IT is empty, the slicing algorithm returns to the second step to find new elements that are relevant to the property $F(st\_Cup = CupIdle)$. This time, the algorithm also adds elements that are already included in the final slice. This guarantees (see Step 5) that their actions should be preserved, i.e. it is forbidden to slice away the actions of these elements.

Iteration 2: Step 2 of the WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{On\}$
since $StandBy \rightarrow_{rcd} On \wedge StandBy \in IS \wedge \neg Refine_R(StandBy)$

$NT = \{t6, t1, t12, t13, t10, t11\}$
since $On \rightarrow_{rdd} t6 \wedge On \in IS \wedge Refine_R(On)$
since $On \rightarrow_{rdd} t1 \wedge On \in IS \wedge Refine_R(On)$
since $On \rightarrow_{rdd} t12 \wedge On \in IS \wedge Refine_R(On)$
since $On \rightarrow_{rdd} t13 \wedge On \in IS \wedge Refine_R(On)$
since $t12 \rightarrow_{sd} t1 \wedge t12 \in IT \wedge Refine_R(t12)$
since $t12 \rightarrow_{tcd} t13 \wedge t12 \in IT \wedge Refine_R(t12)$
since $t12 \rightarrow_{tcd} t10 \wedge t12 \in IT \wedge Refine_R(t12)$
since $t12 \rightarrow_{tcd} t11 \wedge t12 \in IT \wedge Refine_R(t12)$
since $t14 \rightarrow_{sd} t6 \wedge t14 \in IT \wedge \neg Refine_R(t14)$
since $t13 \rightarrow_{sd} t1 \wedge t13 \in IT \wedge \neg Refine_R(t13)$

In the third step, the automaton `Coffee` will be added to the final slice.

Iteration 2: Step 3 of the WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \{CoffeeIdle, CoffeeBusy\}$
$ET = \{t5\}$

This time the fourth step needs a thorough examination. First of all, the state `CoffeeIdle` is added to IS. This is both because the state belongs to ES and because it is an initial state. We have already given an illustration of this in the first iteration. Secondly, the construction of IT may look akward but it is quite logical. Transitions $t11$, $t10$, and $t13$ have to be included in this set because $t12$ depends on them. There actions are going to be preserved in the final slice and therefore it might be possible that these actions reference variables that are defined by states/transitions that are not yet included into the final slice. Such a way of working guarantees the functional equivalence between both models.

Iteration 2: Step 4 of the WDQ-algorithm

4. Reconstruct IS and IT:

$IS = (NS \setminus RS) \cup \{CoffeeIdle\} = \emptyset \cup \{CoffeeIdle\}$

$IT = (NT \setminus RT) \cup \{t11, t10, t13\} \cup \{t5\} = \{t6\} \cup \{t11, t10, t13\} \cup \{t5\}$

Iteration 2: Step 5, 6 and 7 of the WDQ-algorithm

5. $NS \cup ES = \{On\} \cup \{CoffeeIdle, CoffeeBusy\}$
   $NT \cup ET = \{t6, t1, t12, t13, t10, t11\} \cup \{5\}$

$Refine_R(On) = False \wedge On \in NS$
$Refine_R(CoffeeIdle) = False \wedge CoffeeIdle \notin NS$
$Refine_R(CoffeeBusy) = False \wedge CoffeeBusy \notin NS$

$Refine_R(t6) = True \wedge t6 \in NT$
...
$Refine_R(t11) = True \wedge t11 \in NT$
$Refine_R(t5) = False \wedge t5 \notin NT$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES$
$RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

10.1.3 Iteration 3

The job is not done yet. A third iteration is needed.

Iteration 3: Step 2 of the WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{On\}$
since $CoffeeIdle \rightarrow_{rcd} On \wedge CoffeeIdle \in IS \wedge \neg Refine_R(CoffeeIdle)$

$NT = \{t14, t11, t1, t13\}$
since $t10 \rightarrow_{sdd} t14 \wedge t10 \in IT \wedge Refine_R(t10)$
since $t10 \rightarrow_{sdd} t11 \wedge t10 \in IT \wedge Refine_R(t10)$
since $t13 \rightarrow_{sd} t1 \wedge t13 \in IT \wedge Refine_R(t13)$
since $t5 \rightarrow_{sd} t13 \wedge t5 \in IT \wedge \neg Refine_R(t5)$

Iteration 3: Step 3 of the WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \emptyset$
$ET = \emptyset$

Iteration 3: Step 4 of the WDQ-algorithm

4. Reconstruct IS and IT:

$IS = \emptyset$

$IT = (NT \setminus RT) \cup \{t14\} = \{t14\}$

The algorithm continues with the last iteration since IT is not yet empty. We omit to give the details of this last iteration since it only has the responsibility to preserve the action of transition $t14$. The slice returned by the algorithm is the EHA without the `Light` automaton.

10.1.4 Improvements

Table 10.1 shows some improvement results. As can be seen, for this example, the removal of the sequential automaton results in a significant reduction of the used BDD nodes.

Iteration 3: Step 5, 6 and 7 of the WDQ-algorithm

5. $NS \cup ES = \{On\} \cup \emptyset$
   $NT \cup ET = \{t14, t11, t1, t13\} \cup \emptyset$

   $Refine_R(On) = true \wedge On \in NS$

   $Refine_R(t14) = True \wedge t14 \in NT$
   ...
   $Refine_R(t13) = True \wedge t13 \in NT$

6. Reconstruct RS and RT:

   $RS = RS \cup NS \cup ES$
   $RT = RT \cup NT \cup ET$

7. $IS = \emptyset$ and $IT \neq \emptyset$; return to step 2.

Resources	Original Model	Sliced Model
State Variables	33	26
BDD Nodes	29468	10776
User Time	0.10s	0.04s

Table 10.1: Improvements of $F(st\_Cup = CupIdle)$

A binary decision diagram (BDD) is a particular data structure (a graph), used by the model checker CaSMV, which often provides a compact representation of boolean functions. Sets of states of an automaton can be encoded as boolean functions [26] with BDDs . Every BDD has two different types of nodes, terminal nodes and non-terminal nodes. The terminal nodes represent the Boolean values, 0 and 1, while the non-terminal nodes represent variables of the function represented by the BDD i.e. any path from a root node to a terminal node corresponds to a set of variable assignments that make the encoded formula have the terminal node's value.

10.2 Is State `LightOn` Reachable?

Now suppose that we want to verify a trivial property stating that at some point during execution the machine must reach the state `LightOn`. In LTL,

this will be expressed as follows:

$$F(st\_Light = LightOn)$$

The same as before, the slicing criterion extracted from this property will be $< \{LightOn\}, \emptyset >$. That way it is realized that the state `LightOn` is present in the slice. It may be no suprise that this time the slicing algorithm is not able to slice away anything. The sliced statechart is exactly the same as the original model. We may conclude that the WDQ-algorithm only removes elements from the model when it is able to. To be complete, we just give the first iteration of the slicing procedure since the other iterations are almost the same as in Section 10.1.

10.2.1 Iteration 1

Iteration 1: Step 1 of the WDQ-algorithm

1. Given the criterion $< \{LightOn\}, \emptyset >$. Then

$RS = \{LightOn\} \quad Refine_R(LightOn) = True$
$RT = \emptyset$

$ES = \{LightOff\} \quad Refine_R(LightOff) = False$
$ET = \{t16, t17\} \quad Refine_R(t16) = False$
$\qquad\qquad\quad Refine_R(t17) = False$

$RS = RS \cup ES = \{LightOn\} \cup \{LightOff\} = \{LightOn, LightOff\}$
$RT = RT \cup ET = \{t16, t17\}$
$IS = RS = \{LightOn, LightOff\}$
$IT = RT = \{t16, t17\}$

Iteration 1: Step 2 of the WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{On\}$
since $LightOn \rightarrow_{rcd} On \wedge LightOn \in IS \wedge Refine_R(LightOn)$

$NT = \{t6, t12\}$
since $t17 \rightarrow_{sd} t6 \wedge t17 \in IT \wedge \neg Refine_R(t17)$
since $t16 \rightarrow_{sd} t12 \wedge t16 \in IT \wedge \neg Refine_R(t16)$

Iteration 1: Step 3 of the WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \sigma_{Coffee} \cup \sigma_{Controller} \cup (\sigma_{Root} \setminus \{On\})$
$ET = (\delta_{Coffee} \setminus \{t6\}) \cup (\delta_{Controller} \setminus \{t12\}) \cup \delta_{Root}$

Iteration 1: Step 4 of the WDQ-algorithm

4. Reconstruct IS and IT:

$IS = (NS \setminus RS) \cup \{CoffeeIdle, StandBy, Off\} =$
$\{On\} \cup \{CoffeeIdle, StandBy, Off\}$

$IT = (NT \setminus RT) = NT \cup ET = \{t12, t6\} \cup ET$

Iteration 1: Step 5, 6 and 7 of the WDQ-algorithm

5. $NS \cup ES = \{On\} \cup ES$
   $NT \cup ET = \{t12, t6\} \cup ET$

$Refine_R(On) = True \wedge On \in NS$
$Refine_R(s) = False$ if $(s \in ES, s \notin NS)$
$Refine_R(t12) = True \wedge t12 \in NT$
$Refine_R(t) = False$ if $(t \in ET, t \notin NT)$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES$
$RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

10.3 Conclusion

This chapter has clearly shown that the slicing algorithm only slices away some parts of the extended hierarchical automata when they are of no interest. If that is the case, this results in far smaller binary decision diagrams. The smaller the diagram is, the faster the model checker can return its verification result. If parts of the automata are important to the property to be verified, then the algorithm keeps the elements in the final slice. By now, it must be clear that it is not always the case that the slicing algorithm is able to slice away parts of the automata.

CHAPTER 11

Efficient Slicing Theory in Practice

Action expresses priorities.
Mahatma Gandhi.

The WDQ-slicing algorithm, as presented in Chapter 7, has serious shortcomings. In fact, the slicing algorithm is pretty lenient; in order to be certain that the result is still a valid state chart, it keeps some elements that, in fact, could be sliced away. This is especially the case when the hierarchical automata have a lot of synchronization (or broadcasting) issues. However, when synchronization is thoroughly considered during slicing then the returned slice is possibly dramatically smaller than when broadcasting issues are omitted. This is especially relevant to scale finite-state verification techniques.

Chapter 8 has refined some interference dependencies in terms of happens-before relations. These new definitions are concerned with the broadcasting issues that occur in automata. Using these more concrete dependencies during slicing sometimes results in smaller slices (still with respect to the property to be verified), but sometimes it does not. It is the purpose of this chapter to show that the optimized WDQ-algorithm is able to produce smaller slices. To do that, different temporal properties are to be considered again.

11.1 Notations

We use the example of Chapter 8. For clarity reasons, Figure 11.1 shows the same, but annoted EHA.

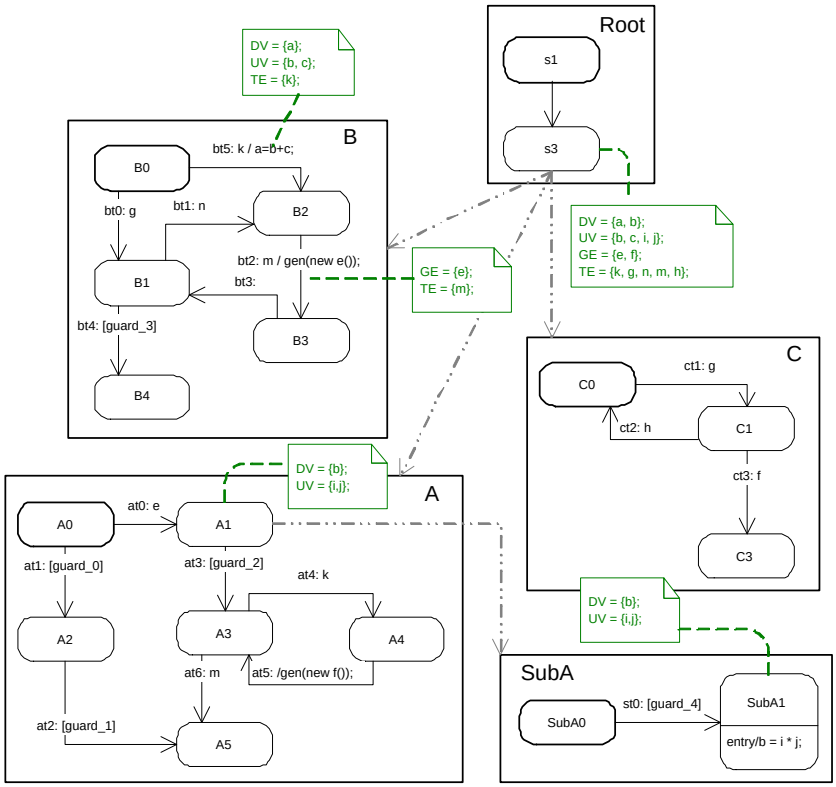


Figure 11.1: Annoted EHA

We define two algorithms: one is referred to as the WDQ-algorithm; the other one is referred to as the OWDQ-algorithm. The *WDQ-algorithm* is the algorithm that does not consider execution chronology. It is the algorithm presented in Chapter 7 and illustrated in Chapter 10. The *OWDQ-algorithm* refers to the optimized algorithm, as discussed in Chapter 8. The only difference with the WDQ-algorithm is that it uses the more precise definitions of some interference dependencies (the ones defined in Section 8.4).

It is the purpose of this chapter to explain the slicing differences between both algorithms in detail. Therefore, we agree that the steps of the WDQ-

algorithm are written in gray, while the steps of the OWDQ-algorithm are written in black.

11.2 Has Variable a the Value 8?

However, suppose that we want to verify a temporal property like this:

$$F(a = 8)$$

Then, the slicing criterion is defined as $< \emptyset, \{bt5\} >$, since bt5 defines the value of the variable a. The reduced model returned by both the original slicing algorithm and the improved one will definitely differ. The WDQ-algorithm only removes automaton C because the algorithm believes that automaton A and SubA compute the value of variable b which is used to define the value of variable a. However, the improved OWDQ-algorithm, which respects the broadcasting mechanism between the regions, removes the automata A, C, and SubA. There is nothing inside these automata that helps region B in defining the value of variable a, or that causes B to make progress. Additionally, the action of transition bt2 is the only one that will be removed from the model, because it is useless to (dis)prove the given property.

We will now show how both algorithms build their slices. As we know from Chapter 8 the OWDQ-algorithm needs to have knowledge of all the happens-before relations that occur in the hierarchical automaton. These relations are used to make some interference dependencies more precise. For simplicity reasons, we omit to give all such relations, but instead only mention the important ones. These are summarized in Table 11.1.

...	...
$B0 \rightarrow_{so} bt5$	$B0 \rightarrow_{so} B2$
$bt5 \rightarrow_{so} B2$	$B0 \rightarrow_{so} bt2$
$bt5 \rightarrow_{so} bt2$	$bt5 \rightarrow_{so} at0$
$bt2 \rightarrow_{so} at0$	$bt2 \rightarrow_{so} A1$
$at0 \rightarrow_{so} A1$	$bt5 \rightarrow_{so} A1$
$at0 \rightarrow_{so} SubA1$	$bt5 \rightarrow_{so} SubA1$
...	...

Table 11.1: Some Happens-Before Relations

11.2.1 Iteration 1

The first step of the algorithm does the initialization of all the sets. Clearly, this is the same for both algorithms.

Iteration 1: Step 1 of the (O)WDQ-algorithm

1. Given the criterion $< \emptyset, \{bt5\} >$. Then

$RS = \emptyset$
$RT = \{bt5\}$ $Refine_R(bt5) = True$

$ES = \sigma_B$ $Refine_R(B0) = False$ $Refine_R(B1) = False$
$Refine_R(B2) = False$ $Refine_R(B3) = False$
$Refine_R(B4) = False$
$ET = \delta_B \setminus \{bt5\}$ $Refine_R(bt0) = False$ $Refine_R(bt1) = False$
$Refine_R(bt2) = False$ $Refine_R(bt3) = False$
$Refine_R(bt4) = False$

$RS = RS \cup ES = \sigma_B$; $RT = RT \cup ET = \delta_B$
$IS = RS$; $IT = RT$

$RS = \emptyset$
$RT = \{bt5\}$ $Refine_R(bt5) = True$

$ES = \sigma_B$ $Refine_R(B0) = False$ $Refine_R(B1) = False$
$Refine_R(B2) = False$ $Refine_R(B3) = False$
$Refine_R(B4) = False$
$ET = \delta_B \setminus \{bt5\}$ $Refine_R(bt0) = False$ $Refine_R(bt1) = False$
$Refine_R(bt2) = False$ $Refine_R(bt3) = False$
$Refine_R(bt4) = False$

$RS = RS \cup ES = \sigma_B$; $RT = RT \cup ET = \delta_B$
$IS = RS$; $IT = RT$

In the second step there is already an important difference noticeable. The WDQ-algorithm will add state `A1` to the final slice due to the fact that `bt5` is parallel data dependent on this state. The OWDQ-algorithm refuses to include this state to the final slice, because it knows that `bt5` never can be parallel data dependent on `A1` since `bt5` happens before `A1` (see Table 11.1).

Iteration 1: Step 2 of the (O)WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{s3, A1\}$ since $B0 \rightarrow_{rcd} s3 \wedge B0 \in IS \wedge \neg Refine_R(B0)$
$bt5 \rightarrow_{pdd} A1 \wedge bt5 \in IT \wedge Refine_R(bt5)$
$NT = \emptyset$

$NS = \{s3\}$ since $B0 \rightarrow_{rcd} s3 \wedge B0 \in IS \wedge \neg Refine_R(B0)$
$NT = \emptyset$

One difference leads to many differences, even in the following steps of the algorithm. In the third step, the WDQ-algorithm adds all the remaining states/transitions of both automata `A` and `Root` to the final slice. Contrarily, the OWDQ-algorithm only adds the remaining elements of the root automaton to the final slice.

Iteration 1: Step 3 of the (O)WDQ-algorithm

3. Reconstruct ES and ET:

$ES = (\sigma_{Root} \setminus \{s3\}) \cup (\sigma_A \setminus \{A1\})$
$ET = \delta_{Root} \cup \delta_A$

$ES = \sigma_{Root} \setminus \{s3\}$
$ET = \delta_{Root}$

The fourth step of the algorithm is easily understood for both versions of the algorithm. It adds those elements to the sets IS and IT that are relevant to the property.

Iteration 1: Step 4 of the (O)WDQ-algorithm

4. Reconstruct IS and IT:

$IS = \{s3, A1\}$
$IT = \emptyset \cup ET = \delta_{Root} \cup \delta_A$

$IS = \{s3\}$
$IT = \emptyset \cup ET = \delta_{Root}$

Iteration 1: Step 5, 6 and 7 of the (O)WDQ-algorithm

5. $NS \cup ES = \sigma_{Root} \cup \sigma_A \quad NT \cup ET = (\delta_{Root} \setminus \{bt5\}) \cup \delta_A$

$Refine_R(s3) = True \wedge s3 \in NS \quad Refine_R(A1) = True \wedge A1 \in NS$
$Refine_R(s \mid s \in ES) = False \quad Refine_R(t \mid t \in ET) = False$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

5. $NS \cup ES = \sigma_{Root} \quad NT \cup ET = (\delta_{Root} \setminus \{bt5\})$

$Refine_R(s3) = True \wedge s3 \in NS$
$Refine_R(s \mid s \in ES) = False \quad Refine_R(t \mid t \in ET) = False$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

11.2.2 Iteration 2

Since the sets IS and IT are not yet empty, the algorithms will go into a second iteration. From this point on, we improve both algorithms with the following:

> *When searching the dependence relations at step 2, if a new state s is found only through refinement control-dependence relation, and there are not any elements depending on s through other relations, then the algorithm will not search the elements which s is refinement data-dependent on in the next iteration when s is reserved in IS.*

With such an improvement, we avoid that possibly irrelevant states are added to the slice. To illustrate this, just look at state `s3`. For both algorithms, `s3` is added to IS due to a refinement control dependence. This way, the algorithms are able to include the root automaton to the slice, and therefore transition $bt5$ is reachable along some execution path.

The only intention of a refinement control dependence is the soundness of the slice. Nothing more, nothing less. Using a refinement data dependence to add for example transition $bt2$ (and its corresponding action) makes no sense; we do not yet know if there is any element dependent on this transition.

Iteration 2: Step 2 of the (O)WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{SubA1\}$ since $A1 \rightarrow_{rdd} SubA1 \wedge A1 \in IS \wedge Refine_R(A1)$
$NT = \{bt2\}$ since $at0 \rightarrow_{sd} bt2 \wedge at0 \in IT \wedge \neg Refine_R(at0)$

$NS = \emptyset$
$NT = \emptyset$

Iteration 2: Step 3 of the (O)WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \sigma_{SubA} \setminus \{SubA1\}$
$ET = \delta_{SubA}$

$ES = \emptyset$
$ET = \emptyset$

Of course, the OWDQ-algorithm is finishing. The final slice returned by the improved algorithm is therefore given by the automata `Root` and `B`. The

Iteration 2: Step 4 of the (O)WDQ-algorithm

4. Reconstruct IS and IT:

$IS = \{SubA1\}$
$IT = \emptyset \cup ET = \delta_{SubA}$

$IS = \emptyset$
$IT = \emptyset$

action of `bt5` is the only action present in the slice. As can be seen, the WDQ-algorithm must go into a third iteration, which will be the last one. Its slice is given by the automata `Root, B, A` and `SubA`. As you and I definitely agree, the slice returned by the OWDQ-algorithm is preferable towards finite-state verification techniques.

Iteration 2: Step 5, 6 and 7 of the (O)WDQ-algorithm

5. $NS \cup ES = \sigma_{SubA} \quad NT \cup ET = \{bt2\} \cup \delta_{SubA}$

$Refine_R(SubA1) = True \wedge SubA1 \in NS$
$Refine_R(s \mid s \in ES) = False$
$Refine_R(bt2) = True \wedge bt2 \in NT \quad Refine_R(t \mid t \in ET) = False$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

5. $NS \cup ES = \emptyset \quad NT \cup ET = \emptyset$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS = \emptyset$ and $IT = \emptyset$; Stop.

11.2.3 Improvements

Table 11.2 shows some important results that speak for themselves; broadcasting may not be neglected while slicing hierarchical automata.

Resources	Original Model	WDQ-Model	OWDQ-Model
State Variables	38	36	26
BDD Nodes	29891	26715	10004
User Time	0.09s	0.07s	0.01

Table 11.2: Improvements of $F(a = 8)$

11.3 Is State B2 Reachable?

Now suppose that we want to verify a trivial property stating that at some point during execution the machine must reach the state B2. In LTL, this will be expressed as follows:

$$F(st\_B = B2)$$

The slicing criterion extracted from this property will be $< \{B2\}, \emptyset >$. That way it is realized that the state B2 is present in the slice. It might be a surprise but the slice returned by both algorithms is exactly the same. Both algorithms will remove the automata A, C, and SubA from the model. This is because the latter regions do not control the execution inside region B in any way. Note that the action of both transitions bt5 and bt2 will be removed as well.

By using this property, we illustrate that using more correct interference dependencies does not always give better results. You will see that during slicing, actions of states and/of transitions are never retained in the final slice. As a direct consequence, the broadcasting information captured in happens-before relations will have no influence on the slicing algorithm.

11.3.1 Iteration 1

The first step of both slicing algorithms is exactly the same. This is because elements are added to the final slice without using dependence information.

Even the second step of both algorithms shows no difference. Both algorithms only add state s3 to the final slice due to a refinement control dependence. Beside B0, none of the elements contained in IS and IT cause new information to be included in the final slice. Therefore, the strength of the OWDQ-algorithm cannot be exploited yet.

Iteration 1: Step 1 of the (O)WDQ-algorithm

1. Given the criterion $< \{B2\}, \emptyset >$. Then

$RS = \{B2\} \quad Refine_R(B2) = True$
$RT = \emptyset$

$ES = \sigma_B \setminus \{B2\} \quad Refine_R(B0) = False \quad Refine_R(B1) = False$
$\qquad Refine_R(B3) = False \quad Refine_R(B4) = False$
$ET = \delta_B \quad Refine_R(bt0) = False \quad Refine_R(bt1) = False$
$\qquad Refine_R(bt2) = False \quad Refine_R(bt3) = False$
$\qquad Refine_R(bt4) = False \quad Refine_R(bt5) = False$

$RS = RS \cup ES = \sigma_B$; $RT = RT \cup ET = \delta_B$
$IS = RS$; $IT = RT$

$RS = \{B2\} \quad Refine_R(B2) = True$
$RT = \emptyset$

$ES = \sigma_B \setminus \{B2\} \quad Refine_R(B0) = False \quad Refine_R(B1) = False$
$\qquad Refine_R(B3) = False \quad Refine_R(B4) = False$
$ET = \delta_B \quad Refine_R(bt0) = False \quad Refine_R(bt1) = False$
$\qquad Refine_R(bt2) = False \quad Refine_R(bt3) = False$
$\qquad Refine_R(bt4) = False \quad Refine_R(bt5) = False$

$RS = RS \cup ES = \sigma_B$; $RT = RT \cup ET = \delta_B$
$IS = RS$; $IT = RT$

Iteration 1: Step 2 of the (O)WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \{s3\}$ since $B0 \rightarrow_{rcd} s3 \wedge B0 \in IS \wedge \neg Refine_R(B0)$
$NT = \emptyset$

$NS = \{s3\}$ since $B0 \rightarrow_{rcd} s3 \wedge B0 \in IS \wedge \neg Refine_R(B0)$
$NT = \emptyset$

To guarantee the soundness of the slice, both algorithms will add the elements

of the root automaton to the slice under construction.

Iteration 1: Step 3 of the (O)WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \sigma_{Root} \setminus \{s3\}$
$ET = \delta_{Root}$

$ES = \sigma_{Root} \setminus \{s3\}$
$ET = \delta_{Root}$

The fourth step of the algorithm is easily understood for both versions of the algorithm. It adds those elements to the sets IS and IT that are relevant to the property.

Iteration 1: Step 4 of the (O)WDQ-algorithm

4. Reconstruct IS and IT:

$IS = \{s3\}$
$IT = \emptyset \cup ET = \delta_{Root}$

$IS = \{s3\}$
$IT = \emptyset \cup ET = \delta_{Root}$

Iteration 1: Step 5, 6 and 7 of the (O)WDQ-algorithm

5. $NS \cup ES = \sigma_{Root} \quad NT \cup ET = \delta_{Root}$

$Refine_R(s3) = True \wedge s3 \in NS$
$Refine_R(s \mid s \in ES) = False \quad Refine_R(t \mid t \in ET) = False$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

5. $NS \cup ES = \sigma_{Root} \quad NT \cup ET = \delta_{Root}$

$Refine_R(s3) = True \wedge s3 \in NS$
$Refine_R(s \mid s \in ES) = False \quad Refine_R(t \mid t \in ET) = False$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS \neq \emptyset$ and $IT \neq \emptyset$; return to step 2.

11.3.2 Iteration 2

Since neither IS nor IT is empty, the slicing algorithms start a second iteration. As you already must guess by now, this second iteration will be the last iteration. Of course, remember that we treat elements that are only added to the slice with a refinement control dependence in a special way. As a consequence, no new information will be added to the final slice.

Iteration 2: Step 2 of the (O)WDQ-algorithm

2. Find new states and transitions using the dependencies.

$NS = \emptyset$
$NT = \emptyset$

$NS = \emptyset$
$NT = \emptyset$

Iteration 2: Step 3 of the (O)WDQ-algorithm

3. Reconstruct ES and ET:

$ES = \emptyset$
$ET = \emptyset$

$ES = \emptyset$
$ET = \emptyset$

Iteration 2: Step 4 of the (O)WDQ-algorithm

4. Reconstruct IS and IT:

$IS = \emptyset$
$IT = \emptyset$

$IS = \emptyset$
$IT = \emptyset$

11.4 Conclusion

Until now, there was no slicing method for concurrent statecharts available which handles some interference dependencies correctly. Interference is introduced through the use of variables in parallel executing regions. Chapter 8 has introduced a more precise definition of these interferences. Using them to slice statecharts with respect to a given property results in more precise slices; and thus in smaller slices. Of course, smaller slices are preferable towards

Iteration 2: Step 5, 6 and 7 of the (O)WDQ-algorithm

5. $NS \cup ES = \emptyset \quad NT \cup ET = \emptyset$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS = \emptyset$ and $IT = \emptyset$; Stop.

5. $NS \cup ES = \emptyset \quad NT \cup ET = \emptyset$

6. Reconstruct RS and RT:

$RS = RS \cup NS \cup ES; \quad RT = RT \cup NT \cup ET$

7. $IS = \emptyset$ and $IT = \emptyset$; Stop.

finite-state verification techniques like model checking. This chapter has illustrated, as more and more statecharts consist of parallel executing regions, that the broadcasting mechanism, as integrated by the OWDQ-algorithm, may not be neglected while slicing statecharts. In fact, the broadcasting mechanism is extremely important to calculate better static slices than other approaches like the WDQ-algorithm.

APPENDIX A

A Final Word

I don't try to describe the future.
I try to prevent it.
Ray Bradbury.

The use of the UML specification language is very widespread due to some of its features. UML models are created during the whole software development process and often several diagrams are used to describe the design of the system with a fair level of detail and long before the implementation of the software is even started. However, the more complex systems of today require modeling methods and tools that allow errors to be detected in the initial phases of development. The use of formal methods makes such error detection possible but the learning cost is high. Indeed, many of these methods require as starting point a formal specification of the software. But not all software developers are familiar with formal specification languages and not all of them are willing to learn or to use them.

This PhD presents a tool which avoids the learning cost, enabling the behavior of a system expressed in UML to be verified in a completely automatic way based on model checking, refinement and slicing. More specifically, the tool automatically carries out a formal framework in which to verify the UML statechart diagrams. The advantage of such an approach is that at the one hand developers can use UML to specify and to describe the software, and at the other hand they can use a "well-known" efficient verification method i.e. model checking.

We have started this research with a translation of an individual statechart into a Kripke model that is used by the model checker CaSMV to prove or to disprove given system properties. To not restrict the complexity of behavioral design specifications, we have also focused on communicating statecharts. This not only augments the complexity of the transformation technique and the corresponding Kripke models, but also influences the practical usefulness of the method (and tool) considerably since quite complex behaviors can automatically be proved to be correct. There is an interesting question to be answered here: how to deal with recursive statecharts? My opinion is that there is a big chance that the mathematics behind model checking needs to be extended.

Additionally, the developed methodology is also capable to deal with what the UML community calls *protocol conformance*. This proves that the verification of statecharts can be done at several levels: at the abstract level of protocols, or at the more detailed behavioral level. To automatically (dis)prove protocol conformance, the model checker is instrumented to use the refinement technique. But one bottleneck is present here: the model checker CaSMV forces us to use a very strict refinement function – a one-to-one mapping- to relate behavioral statecharts to protocol statecharts. And this, of course, limits capabilities. As a side effect, both types of statecharts are syntactically very close. To overcome such a limitation, it is interesting to investigate how to adapt the model checker CaSMV in such a way that more complex refinement functions are allowed or that could be defined by the user. To go even further in the field of mathematics, some maybe find it attractive to find a connection with process algebraic techniques and concepts.

The traditional problem of formal verification like model checking is the so-called *state space explosion* problem, which makes the verification of huge UML statechart models infeasible. Applying the slicing technique to a statechart model provides a nice work-around to this problem; slicing removes pieces of the model that are not of interest during verification. This research mentions a nice optimization of the slicing technique. The optimization takes into account the broadcasting mechanism between different parts of a statechart model to remove false dependencies between parts of the statechart. As a result, the complexity of the verification approach is even more reduced. In the future, we would like to develop an elegant slicing algorithm for models consisting of communicating statecharts. We are also planning to flood the slicing algorithm with industrial strength examples.

We hope that our verification tool will contribute to improve the quality of UML models. In fact, it is possible as our practical examples clearly illustrate. However, UML is a rich language and its study and formalization goes beyond the effort of a single person or research team. This is one of the reasons why we have focused our attention to statechart diagrams only. In the long run, there will be (there already are) many different verification and validation methods and tools for many different subsets of the UML formalism. It is up to the software developer to freely choose the method or methods that better suit his model and the problem that needs to be solved. Our tool is then chosen at the moment the system is required to have a correct behavior at all times. At the end, the last step is to provide a single tool that has all the knowledge needed to analyze the different diagrams of software design, and that can be installed in the workstation of any UML designer.

APPENDIX B

Nederlandstalige Samenvatting

"I don't know"
Wat betekent dat in het Nederlands?
"Sorry, dat weet ik niet".
W. van Broeckhoven.

B.1 Algemeen

Computers hebben een grote impact gehad op ons leven, al vanaf het moment dat ze uitgevonden zijn. Nu maken ze integraal deel uit van ons dagelijks leven. Denk bijvoorbeeld aan keukenapparatuur, medische toestellen, verkeerssystemen, ...; het zijn allemaal computers. Doordat onze afhankelijkheid van computers in de toekomst zal blijven toenemen, is het van groot belang dat we van dergelijke systemen kunnen aantonen dat ze foutloos werken. We hebben immers dringend nood aan applicaties die in hoge mate betrouwbaar zijn. *Hoe bekomen we nu zulke applicaties?* Dit is de centrale vraag waarop dit proefschrift een antwoord tracht te geven, maar wel voor een specifieke groep van systemen, namelijk *ingebedde systemen*[1].

B.1.1 Ingebedde Systemen

Ingebedde systemen zijn computers die we terugvinden in de meest uiteenlopende producten: auto's, wasmachines, mobiele telefoons, etc. Van buitenaf is het niet direct te zien dat er inwendig van een computer gebruik gemaakt wordt. Ingebedde systemen breiden de functionaliteit van een

[1]Het onderzoek uiteengezet in dit proefschrift kan ook gebruikt worden voor andere applicaties.

apparaat uit waardoor de intelligentie van het product verhoogt. Bijvoorbeeld, wanneer een broodmachine uitgebreid wordt met een elektronische tijdsschakelaar (brood wordt gebakken tegen de ochtend aan), dan is die tijdsschakelaar het ingebedde systeem. De uitbreiding van de functionaliteit wordt doorgaans in hoge mate bepaald door de toepassing van software.

Ingebedde systemen zijn duidelijk verschillend van standaard computers. Om het belangrijkste verschil te noemen, ieder ingebed systeem heeft een specifieke taak, wat niets anders is dan een functionaliteit naar de gebruiker toe. Onze desktops daarentegen kunnen voor heel veel verschillende taken ingezet worden. Om nog een ander verschil te noemen, een ingebed systeem is meestal beperkt in zijn resources (geheugen etc.) en heeft een volledige andere gebruikersinterface dan standaard computers. Het toetsenbord is vervangen door knopjes waarvan er niet meer zijn dan nodig en de monitor is vervangen door een eenvoudig display, een venstertje waar je informatie af kunt lezen die voor het systeem belangrijk is.

B.1.2 Máák een Ontwerp

Nog steeds beweren sommige software-ingenieurs dat ze tijd besparen door het ontwerpen van een programma over te slaan. Het *ontwerp*[2] van nieuwe software begint dan als een mooi model in het brein van de programmeur en wordt dan ook onmiddellijk gemaakt in een programmeertaal. In dit stadium is het zuiver, elegant en onweerstaanbaar. Na de eerste release gaat het mis. De software begint te "rotten". In eerste instantie is het niet zo'n probleem. Een lelijke patch hier, een onhandige aanpassing daar, maar de schoonheid van het ontwerp is nog steeds zichtbaar. In de loop van de tijd tasten alle aanpassingen het ontwerp echter aan. Het wordt spaghetticode, dat uiteraard steeds moeilijker te onderhouden is (uit [84]). Het duurt uren en zelfs weken om de kwaliteit en de betrouwbaarheid van het product te verbeteren.

Toch bestaat er een zeer eenvoudige oplossing voor dit probleem: indien je eerst het product ontwerpt in een menselijke taal, dan duurt het maar enkele minuten om verbeteringen aan te brengen. Pas wanneer het ontwerp compleet is en aanvaard wordt, kan de implementatie van het product starten. Dit maakt dat een ontwerp een vereenvoudiging is van de realiteit; het is een blauwdruk van het eigenlijke systeem. Ieder ontwerp, ook wel *model* genoemd, zorgt ervoor dat softwareontwikkelaars het systeem beter begrijpen en het product beter kunnen implementeren, aanpassen of uitbreiden. Met behulp van een ontwerp kunnen beslissingen heel snel en efficiënt gewijzigd worden, zelfs op implementatieniveau.

[2]Het softwareontwerp kan je het beste vergelijken met het architectenplan van je huis.

Ingebedde systemen zijn meestal vrij complexe systemen waaraan steeds hogere eisen gesteld worden betreffende onder meer de reactietijd, de beschikbaarheid, de veiligheid, de robuustheid, de integreerbaarheid en de onderhoudbaarheid. Voor deze systemen is het uitermate belangrijk dat de implementatiefase voorafgegaan wordt door een zeer nauwkeurige objectgeoriënteerde ontwerpfase.

B.1.3 Ontwerpen met UML

De Unified Modeling Language (UML) [96] is de standaardtaal die gebruikt wordt tijdens de objectgeoriënteerde analyse- en ontwerpfase van softwareprojecten. De taal is geschikt voor het maken van modellen van de software. Deze modellen worden vervolgens ofwel geheel handmatig ofwel gedeeltelijk automatisch naar programmacode omgezet.

Met behulp van UML kunnen software-ingenieurs zowel statische als dynamische aspecten van een systeem beschrijven. Hiertoe onderscheidt UML verschillende soorten diagrammen. Zo kan met behulp van klassediagrammen de architectuur van een systeem gespecificeerd worden: de klassen, de relaties tussen de klassen, de interfaces, ... Een statechartdiagram laat toe om de status weer te geven waarin een object zich kan bevinden tijdens zijn bestaan in het systeem. Het toont ook de overgangen tussen de verschillende toestanden samen met de events en de activiteiten die deze veranderingen in de toestanden veroorzaken.

UML-modellen worden gebouwd en gebruikt tijdens het gehele softwareontwikkelingsproces. Software-ingenieurs kunnen UML-diagrammen opstellen tijdens de initiële ontwerpfases, zelfs wanneer het nog niet volledig duidelijk is hoe het systeem er uiteindelijk zal uitzien. Naarmate de ontwikkeling evolueert, worden de diagrammen aangepast, worden ze gedetailleerder of verschijnen er nieuwe diagrammen. Dit proces herhaalt zich tot wanneer het gehele team tevreden is met het ontwerp. Vervolgens wordt de software geïmplementeerd overeenkomstig dit model. De kwaliteit van de software hangt dus af van de kwaliteit van het model.

B.1.4 De Nood aan Verificatie

Ingebedde systemen maken hoe langer hoe meer deel uit van ons dagelijks leven. De nood aan correctheidsgaranties wordt echter alsmaar groter aangezien we te veel afhankelijk geworden zijn van hun continue en correcte werking. Denk bijvoorbeeld aan de gevolgen van een ontwerpfout in de besturingssoftware van een vliegtuig of trein. Software maken zonder verificatie wordt daarom ondenkbaar. Om nu aan de vraag naar grotere

kwaliteit te voldoen is een wezenlijke verbetering van het ontwerpproces nodig. Fouten moeten zo vroeg mogelijk in het ontwikkelingsproces van het systeem verwijderd worden.

Elk ontwikkelingsproces start met een ontwerpfase die de volgende fase, de implementatiefase, veel vlotter doet verlopen. Echter, ingebedde systemen zijn complexe systemen wat betekent dat ook het softwareontwerp steeds complexer wordt waardoor het zo goed als onmogelijk is om manueel na te gaan of zo een systeem wel degelijk aan de gestelde eisen voldoet. Ook zijn het aantal denkbare situaties dermate groot dat testen onvoldoende zekerheid geven over de correctheid. Het is immers mogelijk dat het ontwerp een gedrag bevat dat wij, softwareontwikkelaars, niet verwachten. Dit gedrag kan zware fouten in het systeem veroorzaken. Het opsporen van fouten aan de hand van bewijsvoeringsmethodes moet daarom ook gebeuren in de ontwerpfase en niet alleen in de implementatiefase. Meer nog, de kost om fouten te vinden en te verwijderen groeit drastisch naarmate ze de implementatiefase binnengedrongen zijn.

Laten we nu een voorbeeld bekijken waaruit blijkt dat verificatie wel degelijk onmisbaar geworden is in het ontwikkelingsproces van software. Een beroemd voorbeeld betreft de eerste Ariane 5 raket (vlucht 501) die in 1996 kort na lancering door een softwarefout, en noodgedwongen tot ontploffing gebracht werd. Gelukkig betrof dit een onbemand tuig. Het was enkel door toepassing van formele verificatie dat de oorzaak van deze fout gedetecteerd werd. Er trad een overloop (overflow) en de computer werd uitgeschakeld. De overloop (overflow) werd veroorzaakt door een data conversie van een 64-bit floating point naar een 16-bit signed integer waarde. Het floating point getal had een waarde die veel te groot was voor de voorstelling als 16-bit signed integer. Dit voorbeeld illustreert duidelijk dat indien eerst verificatie wordt toegepast, dergelijke situaties vermeden kunnen worden. Het spaart veel tijd, veel geld, en soms ook wel mensenlevens.

B.2 Verificatiemethode

Ingebedde systemen zijn eigenlijk interactieve computersystemen. De inputs worden doorlopend van de buitenwereld verkregen, terwijl de outputs steeds aan de buitenwereld ter beschikking worden gesteld. Eenvoudiger uitgelegd, ingebedde systemen zijn reactieve systemen; ze wachten op gebeurtenissen waarmee ze wat moeten doen, of nog, hun acties worden veroorzaakt door reacties op externe events.

Voor deze ingebedde systemen is de toepasbaarheid van het “state machine” gedachtegoed bijna vanzelfsprekend. De modellering en de im-

plementatie van dergelijke systemen vallen zeer natuurlijk te omschrijven in termen van toestandmachines (UML-statecharts). Het zijn formele modellen waarmee het gedrag van het ingebedde systeem als een reeks van overgangen van de ene toestand naar de andere beschreven wordt. Figuur B.1 beschrijft het vanzelfsprekende gedrag (sterk vereenvoudigd) van een koffiezetautomaat met behulp van een UML-statechart.

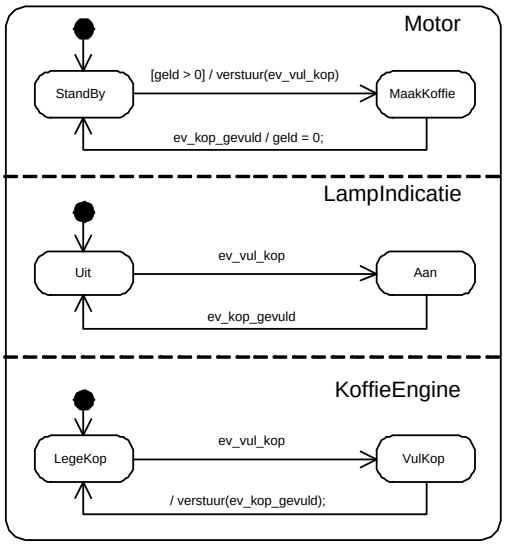


Figure B.1: Het Gedrag van een Koffiezetautomaat

Het ontwerp (en de implementatie) van ingebedde systemen is zeer gecompliceerd geworden, en daardoor uitermate gevoelig voor fouten. Voor een klein model kan de correctheid van een ontwerp met de hand bewezen worden. Doch kan dit soms een tijdrovende en foutgevoelige bezigheid zijn. Traditionele simulaties (testen) zullen ook tekortschieten in het blootleggen van fouten in het ontwerp. Daarom is het beter gebruik te maken van geautomatiseerde formele methoden voor de verificatie.

De gerenommeerde formele verificatietechniek *model checking* is een aanvullende methode die zo goed als geen expertise vereist; het is als het ware een "simulatie in een hogere versnelling". Model checking controleert het systeem op fouten als volgt: er wordt steeds gestart bij de beginsituatie van het systeem en vervolgens worden alle paden die het systeem kan nemen automatisch afgelopen. Met behulp van model checking kunnen we bijvoorbeeld nagaan dat het systeem nooit in een fouttoestand kan belanden. Indien deze fouttoestand wel bereikt kan worden, dan kan het model checking

algoritme ook aanduiden via welk pad (= tegenvoorbeeld) deze fouttoestand bereikt kan worden. Dergelijke informatie is belangrijk om na te gaan waar de fout in het ontwerp zit. De fout in het ontwerp wordt vervolgens aangepast waardoor een betrouwbaarder en veiliger systeem ontstaat. Het moet duidelijk zijn dat model checking gebaseerd is op een automatische en een uitputtende verkenning van de bereikbare toestandsruimte van het systeem. Deze thesis maakt gebruik van de model checker Cadence SMV (CaSMV) [86] om zo de kwaliteit van ingebedde systemen te verhogen.

Aan de ene kant hebben we UML-statecharts, aan de andere kant hebben we de model checker CaSMV die we kunnen inzetten om na te gaan of het systeem al dan niet aan de gegeven eigenschappen voldoet (zie Figuur B.2). Een belangrijke bijdrage van deze thesis is de transformatie van het grafische statechartmodel naar het textuele inputformaat van CaSMV. De volledige uitvoeringssemantiek van statecharts mag niet verloren gaan tijdens deze transformatie. Daartoe hebben we eerst de uitvoeringssemantiek formeel gedefinieerd om zo het transformatieproces vlotter te laten verlopen (zie hiervoor Hoofdstuk 2).

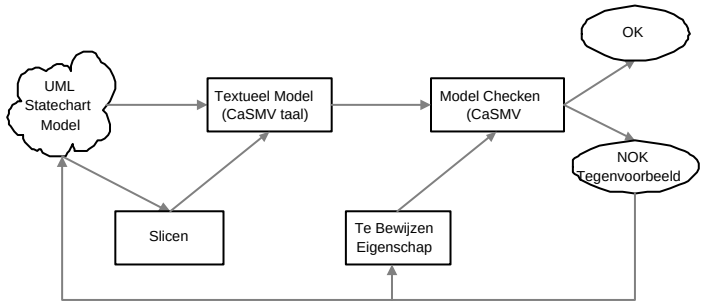


Figure B.2: Integratie van Model Checking in het Ontwerpproces

De gevolgde methode is in de literatuur niet onbekend. Toch vinden wij dat de voorgestelde methodes fundamentele tekorten bevatten. Ten eerste bevat de literatuur een eerste klasse van methodes die enkel in staat zijn om ontwerpen bestaande uit juist één UML statechart te verifiëren. Deze methodes maken voor de verificatie ook gebruik van een backend-tool zoals SPIN [55], of CaSMV [86]. Dergelijke ontwerpen zijn voor ingebedde systemen echter niet altijd even realistisch. Uiteraard is het wel zo dat een initieel ontwerp van een ingebed systeem start met een enkele statechart maar al snel zullen meerdere statecharts aan het ontwerp toegevoegd worden, zoals uit latere voorbeelden zal blijken. Bovendien bleek na onderzoek

ook dat de methodes een onvolledige en/of een onnauwkeurige en/of een beperkte uitvoeringssemantiek integreerden in het verificatieproces. Dit legt niet alleen modelleringsbeperkingen op aan software-ingenieurs, maar zorgt er ook voor dat de gevolgde semantiek tijdens de verificatie verschilt van de semantiek die gevolgd wordt tijdens de implementatie. Niet verwonderlijk dat dit nadelig is voor zowel de kwaliteit als de betrouwbaarheid van het systeem. De semantiek van één statechart die wij gedefinieerd hebben in Hoofdstuk 2, werkt deze nadelen weg en ligt aan de basis van het gehele transformatieproces. In Hoofdstuk 3 hebben wij het transformatieproces van een enkel statechart nauwkeurig beschreven. Zowel de semantiek als het transformatieproces zijn zodanig opgebouwd dat ze zich eenvoudig laten uitbreiden naar meer realistische ontwerpen van ingebedde systemen.

Anderzijds kent de literatuur ook een methode die reeds complexere ontwerpen kan verifiëren op de juistheid van eigenschappen. Deze ontwerpen bevatten meerdere UML-statecharts met de veronderstelling dat ze altijd parallel ten opzichte van elkaar werken. Voor ingebedde systemen is dit al meer aanvaardbaar. Laten we, ter illustratie, terugkeren naar het voorbeeld van de koffiezetautomaat. Een koffiezetautomaat reageert op inputs van de omgeving. Bijvoorbeeld, de automaat biedt koffie aan nadat een klant het juiste geldbedrag in de machine heeft geworpen. Uiteraard zijn zowel de automaat als de klant autonome entiteiten die tegelijkertijd (= in parallel) acties kunnen uitvoeren. Indien we nu een dergelijk systeem wensen te ontwerpen, ontstaat een model bestaande uit verschillende concurrente statecharts; het gedrag van het koffiezetapparaat wordt beschreven in een statechart en het gedrag van de klant eveneens. Indien ook de onderhoudsman aan het systeem wordt toegevoegd, dan kan zijn gedrag eveneens gespecificeerd worden in nog een andere statechart. Maar een ingebed systeem kan inwendig ook bestaan uit verschillende concurrente componenten. Het gedrag van elke component wordt dan gespecificeerd in afzonderlijke, doch parallele statecharts. Denk bijvoorbeeld aan de verschillende onderdelen van je auto.

Maar we zijn nog niet tevreden, we willen een nog grotere groep van ontwerpen verifiëren, net daar waar de literatuur afhaakt. We willen eigenschappen kunnen bewijzen over ontwerpen bestaande uit zowel *actieve* als *passieve objecten*. Actieve objecten zijn de zogenaamde autonome concurrente entiteiten waarover we zonet gesproken hebben. Dit in tegenstelling tot passieve objecten; zij kunnen niet uitvoeren in parallel met andere objecten. Zo is een wasmachine een actief object (het staat altijd aan, of het voert uit), maar de onderdelen (watertank, sproeier, droger) ervan zijn passieve objecten die nooit tegelijkertijd actief zullen zijn. Hoofdstuk 4 beschrijft de semantiek die we nodig hebben voor dergelijke ontwerpen. Het

is immers belangrijk te weten hoe actieve en passieve objecten met elkaar communiceren, welke events onmiddellijk afgehandeld moeten worden en welke niet, enz. Deze semantiek is in feite een zeer eenvoudige uitbreiding van de semantiek gedefinieerd in Hoofdstuk 3. Deze semantiek zal opnieuw een zeer grote rol spelen tijdens het transformatieproces van dit soort ontwerpen.

Samengevat hebben we een methode ontwikkeld die in staat is een grote diversiteit aan ontwerpen (juist één statechart, enkel actieve statecharts, zowel actieve als passieve statecharts) te verifiëren op hun correctheid. Enkel en alleen op die manier kunnen we de kwaliteit van complexe ingebedde systemen verhogen aangezien fouten in complexe ontwerpen vroegtijdig gesignaleerd kunnen worden.

Uiteraard hebben we ook nood aan een logisch formalisme waarmee de correctheidseigenschappen op een formele manier uitgedrukt kunnen worden. Dit wordt beschreven in hoofdstuk 5.

We gaan nog een stapje verder. Als je vertrouwd bent met objectgeoriënteerde methodes, dan weet je dat er zoiets bestaat als klassen en interfaces. Een *klasse* is de abstractie van een entiteit uit een bepaald systeem. Iedere klasse omvat een gedeelte van de systeemfunctionaliteit. Het is een stukje code dat beschrijft hoe de entiteit er in werkelijkheid uitziet, wat het kan doen en hoe alles werkt. Een *interface* daarentegen is een variant van een klasse. Interfaces maken het mogelijk een onderscheid te maken tussen wat een object kan en zijn daadwerkelijke implementatie. Een interface kan dus worden geïmplementeerd door twee verschillende klassen die helemaal niets met elkaar te maken hebben, echter door de interface kunnen ze wel uniform benaderd worden.

De laatste nieuwe versie van de UML standaard laat nu toe dat het gedrag van klassen gespecificeerd wordt aan de hand van zogenaamde *gedragsstatecharts*, dewelke dezelfde zijn als de statecharts zoals we ze tot nu toe gebruikt hebben. Anderzijds bestaan er ook *protocol statecharts* die enkel aan interfaces gekoppeld mogen worden. Met behulp van deze statecharts kunnen we aanduiden hoe een klasse de interface moet gebruiken; welke events in welke volgorde mogen optreden; dus op welke manier een klasse dient te reageren op inputs. De acties die gebeuren ten gevolge van deze reacties worden *niet* weergegeven.

Wat is nu het probleem? Enerzijds hebben we een gedragsstatechart, anderzijds hebben we een protocol statechart, hoe kunnen we nu bewijzen dat een klasse de opgelegde regels van zijn interface wel degelijk implementeert? Dit noemen we *protocol conformance verificatie* en wordt uitvoerig behandeld in Hoofdstuk 6. De verificatie is zodanig gemodelleerd dat ze combineerbaar is met de verificatiemethode uit Hoofdstukken 3-4. Bovendien is het

zo dat het consistentiebewijs tussen de twee soorten statecharts volledig automatisch uitgevoerd kan worden. Een groot plusplunt! De kwaliteit van de systemen wordt opnieuw verhoogd.

B.3 Optimalisatiemethode

Zoals we reeds vermeld hebben, is het niet onwaarschijnlijk dat het software-ontwerp dermate complex is dat het nodig is om speciale aandacht te besteden aan de correctheid ervan. Dit kan gerealiseerd worden door gebruik te maken van model checking, wat een geautomatiseerde formele verificatiemethode is, en waarmee zowel gewenste als ongewenste eigenschappen bewezen of weerlegd kunnen worden. Het probleem waar model checking altijd mee te kampen heeft, is het probleem van de toestandsexplosie: het aantal toestanden van een systeem kan exponentieel groeien met het aantal componenten van een systeem, waardoor de grootte van de systemen die daadwerkelijk geverifieerd kunnen worden ernstig beperkt wordt. Anders gezegd, er ligt een beperking op de grootte van de statecharts die gebruikt worden om het systeemgedrag te specifieren in de ontwerpsfase. Hoe groter de statecharts, hoe groter de kans dat tijdens de verificatie de model checker *out-of-memory* gaat.

Maar niet getreurd, voor elk probleem bestaat een oplossing. De literatuur beschikt reeds over een algoritme dat het probleem van de toestandsexplosie reduceert (besproken in Hoofdstuk 7). Dit algoritme is gebaseerd op de program slicing techniek. Het slicing algoritme filtert als het ware uit een statechart een veel kleinere statechart. En het is nu net dit kleinere statechart waarop de model checker een systeemeigenschap gaat bewijzen of weerleggen. Uiteraard bevat de kleinere statechart alle informatie die relevant is voor een bepaalde te bewijzen eigenschap. De irrelevante informatie wordt met andere woorden achterwege gelaten.

Laten we het slicing idee even illustreren op het voorbeeld van de koffiezetautomaat. Stel dat we de volgende eigenschap willen nagaan: *van zodra het event* `ev_vul_kop` *gegenereerd wordt, moet de machine in de toestand* `VulKop` *terechtkomen.* Om deze eigenschap te bewijzen is het niet nodig dat het volledige statechart als input naar de model checker gestuurd wordt. De toestand `LampIndicatie` kan probleemloos uit dit model verwijderd worden, zonder te raken aan de waarheidswaarde van de eigenschap. Dit is omdat deze toestand slechts de verwerkingsindicatie geeft naar de gebruiker toe. Het bekomen van een kleiner model door wegwerking van dergelijke irrelevante toestanden is nu net de verantwoordelijkheid van het bestaande statechart slicing algoritme.

Uiteraard bevat het bestaande slicing algoritme een belangrijke tekortkoming. Het houdt te weinig rekening met de concurrente aspecten binnenin de statecharts. Het is nu net de doelstelling van Hoofdstuk 8 om deze concurrente aspecten te verwerken in het bestaande slicing algoritme. Op die manier kunnen we nog meer irrelevante informatie verwijderen met betrekking tot een te bewijzen eigenschap. Er worden dus als het ware nog kleinere modellen gefilterd uit de originele statecharts.

Wat is nu het grote plusplunt aan het slicing algoritme? Wel, we creëren de mogelijkheid om grotere modellen te bewijzen op de juistheid ervan, door ons enkel te richten op een relevant deelmodel. Voor de model checker betekent dit dat hij minder te kampen krijgt met het probleem van de explosie van de toestandsruimte.

B.4 Besluit

Dit proefschrift beschrijft een onderzoek op het gebied van software verificatie waarbij enkel gekeken wordt naar de dynamische aspecten ervan. De nadruk ligt op de verificatie van het ontwerp van ingebedde systemen om zo de betrouwbaarheid van dergelijke systemen te verhogen. De verificatie maakt gebruik van een gerenommeerde bewijsvoeringsmethode, *model checking*, die gebruikt wordt in zowel de academische wereld als de bedrijfswereld. Aangezien model checking gebaseerd is op het onderzoek van toestandsruimtes, is het struikelblok voor de toepasbaarheid ervan het probleem van de explosie van deze toestandsruimtes. In dit proefschrift hebben we een bestaande techniek geoptimaliseerd om dit probleem te verlichten.

Bibliography

[1] *Special Issues on Embedded Systems*, volume 9. IEEE Computer, 2000.

[2] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software Inspections: An Effective Verification Process. *IEEE Softw.*, 6(3):31–36, 1989.

[3] Sandeep Agrawal and Pankaj Bhatt. *Real-Time Embedded Software Systems, An Introduction*, Technology Review 2001. Version 1.0.

[4] Carina Andersson. Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection. Master's thesis, Lund University, November 2003.

[5] Demissie B. Aredo. Semantics of UML Sequence Diagrams in PVS.

[6] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[7] André Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International (UK) Ltd., 1994. Translator-John Plaice.

[8] CT Arrington. *Enterprise Java with UML*. Wiley & Sons, 2001.

[9] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. Style-Based Refinement of Dynamic Software Architectures. In *Proc. WICSA 2004: 4th Working International IEEE/IFIP Conference on Software Architecture*, pages 155–164, Oslo, Norway, 2004. IEEE Computer Society.

[10] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente. UML Automatic Verification Tool with Formal Methods.

[11] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The Temporal Logic Sugar. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001.

[12] B. Berard, M. Bidot, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie, editors. *Systems and Software Verification.* Springer, 2001.

[13] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and Data-flow Analysis of While-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.

[14] Marco Bernardo and Flavio Corradini. On the Semantic Foundations of Standard UML 2.0. *Lect. Notes Comp. Sci.*, 3185:181–199, 2004.

[15] Purandar Bhaduri and S. Ramesh. Model Checking of Statechart Models: Survey and Research Directions. *CoRR*, cs.SE/0407038, 2004.

[16] David Binkley and Keith Brian Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[17] Paul. E. Black. Kripke Structure, from Dictionary of Algorithms and Data Structures, Paul E. Black, ed., NIST.

[18] Andrea Bondavalli, Mario Dal Cin, Diego Latella, István Majzik, András Pataricza, and Giancarlo Savoia. Dependability Analysis in the Early Phases of UML-Based System Design. *Comput. Syst. Sci. Eng.*, 16(5):265–275, 2001.

[19] E. Börger and L. Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5):603–665, May 1997. `http://www.jucs.org/jucs_3_5/integrating_asm`.

[20] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Softw.*, 12(4):34–41, 1995.

[21] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments Revisited: a Ten-Year Perspective on the Industrial Application of Formal Methods. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 8–16, New York, NY, USA, 2005. ACM Press.

[22] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theor. Comput. Sci.*, 59:115–131, 1988.

[23] Tanuan M. C. Automated Analysis of Unified Modeling Language (UML) Specifications. Master's thesis, University of Waterloo, 2001.

[24] T. Cattel. Process Control Design using Spin, 1995.

[25] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. *IEEE Trans. Softw. Eng.*, 24(7):498–520, 1998.

[26] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2002. 0-262-03270-8.

[27] E. M. Clarke and W. Heinle. Modular Translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, August 2000.

[28] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Form. Methods Syst. Des.*, 6(2):217–232, 1995.

[29] Jude Community. Jude. Available from `http://objectclub.esm.co.jp/Jude/`.

[30] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[31] Michelle L. Crane and Juergen Dingel. On the Semantics of UML State Machines:Categorization and Comparison. Technical report, School of Computing Queen's University Kingston, Ontario, Canada, 2005.

[32] György Csertán, Mario Dal Cin, Gábor Huszerl, Judit Jávorszky, Kostas Kosmidis, András Pataricza, and Csaba Szász. The Demonstrator. Technical Report HIDE/D5/TUB/1/v2, ESPRIT Project 27439 (HIDE, 1999.

[33] Jeffrey Dallien and Wendy MacCaull. Automated Checking for Stutter Invariance of LTL Formulas. In *Proceedings of the 28th Annual APICS Conference*, 2004.

[34] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

[35] Adm Darvas and Istvn Majzik. Verification of UML Statechart Models of Embedded Systems. In *Proc. 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS 2002)*, pages 70–77, 2002.

[36] Alexandre David, Johann Deneux, and Julien d'Orso. A Formal Semantics for UML Statecharts. Technical Report 2003-010, Uppsala University, 2003.

[37] Bruce Powel Douglass. UML Statecharts. White Paper.

[38] Bruce Powel Douglass. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 2000.

[39] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[40] Cindy Eisner, Dana Fisman, John Havlicek, Anthony McIsaac, and David Van Campenhout. The Definition of a Temporal Clock Osperator. In *Proc. 30th Int. Colloq. Aut. Lang. Prog.*, LNCS 2719, pages 857–870. Springer, June 2003.

[41] Martin Fränzle. High-Level Modeling with Statecharts. Course at the Technical Univerisity of Denmark, 2003.

[42] Peter Freeman and David Hart. A Science of Design for Software-intensive Systems. *Commun. ACM*, 47(8):19–21, 2004.

[43] Vinod Ganapathy and S. Ramesh. Slicing Synchronous Reactive Programs. *Electr. Notes Theor. Comput. Sci.*, 65(5), 2002.

[44] Sébastien Gérard and Ileana Ober. Parallelism/Concurrency Specification within UML. White Paper.

[45] James Gleick. A Bug and a Crash. New York Times Magazine (1 December 1996).

[46] Stefania Gnesi, Diego Latella, and Mieke Massink. Model Checking UML Statechart Diagrams Using Jack. In *HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 46–55, Washington, DC, USA, 1999. IEEE Computer Society.

[47] Martin Gogolla. Graph Transformations on the UML Metamodel. In Jose D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[48] D. Harel and H. Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *Lect. Notes in Comp. Sci.*, pages 325–354. Springer-Verlag, 2004.

[49] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[50] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.

[51] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[52] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and Slicing of Hierarchical State Machines. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 450–467. Springer–Verlag, 1997.

[53] Hans Hendrickx. Automatische Generatie van een Formeel Model voor UML Statecharts. Master's thesis, Universiteit Gent, 2006.

[54] Gerard J. Holzmann. ON-THE-FLY, LTL MODEL CHECKING with SPIN. Available from `http://spinroot.com/spin/whatispin.html`.

[55] Gerard J. Holzmann. *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional, 2004.

[56] Hyoung Seok Hong, Jeong Hyun Kim, Sung Deok Cha, and Yong Rae Kwon. Static Semantics and Priority Schemes for Statecharts. In *Computer Software and Applications Conference (COMPSAC) '95*, pages 114–120, 95.

[57] IBM. Rational Software. Available from `http://www-306.ibm.com/software/rational/`.

[58] iLogix. Rhapsody. Available from `http://www.ilogix.com/`.

[59] Tenzer J. and Stevens P. Modelling Recursive Calls with UML State Diagrams. *Proceedings of Fundamental Approaches to Software Engineering, LNCS*, 2621, 2003.

[60] Sebastian John. Transition Selection Algorithms for Statecharts. In *GI Jahrestagung (1)*, pages 622–627, 2001.

[61] S. Kent, A. Evans, and B. Rumpe. Uml Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*. LNCS 1743, Springer Verlag, 1999.

[62] Alexander Knapp, Stephan Merz, Martin Wirsing, and Júlia Zappe. Specification and Refinement of Mobile Systems in MTLA and Mobile UML. *Theoretical Computer Science*, 2005. Special Issue AMAST 2004.

[63] B. Korel and J. Laski. Dynamic Program Slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[64] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Joh. Kepler University Linz, September 2000. `http://www.gup.uni-linz.ac.at/~dk/thesis`.

[65] Jens Krinke. Static Slicing of Threaded Programs. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42, New York, NY, USA, 1998. ACM Press.

[66] Jens Krinke. Context-sensitive Slicing of Concurrent Programs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[67] Gihwon Kwon. Rewrite rules and Operational Semantics for Model Checking UML Statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[68] Yassine Lakhnech, Erich Mikk, and Michael Siegel. Hierarchical Automata as Model for Statecharts. In *Proc. of the Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer-Verlag, 1997.

[69] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[70] Leslie Lamport. What Good is Temporal Logic? In *IFIP Congress*, pages 657–668, 1983.

[71] Gérard Le Lann. An Analysis of the Ariane 5 Flight 501 Failure-a System Engineering Perspective. In *ECBS*, pages 339–246, 1997.

[72] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[73] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of*

the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), page 465. Kluwer, B.V., 1999.

[74] E. Lee. Embedded Software, 2002.

[75] Edward A. Lee. Embedded Software. volume 56. Academic Press, London, 2002.

[76] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, London, UK, 1995. Springer-Verlag.

[77] Johan Lilius and Ivan Paltor. vUML: A Tool for Verifying UML Models. In *ASE*, pages 255–258, 1999.

[78] Johan Lilius and Ivan Paltor. The Production Cell: An Exercise in the Formal Verification of a UML Model. In *HICSS*, 2000.

[79] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, Inc., 2001.

[80] Jacques-Louis Lions et al. Ariane 5 Flight 501 Failure Report by the Inquiry Board. Technical report, European Space Agency, Paris, France, 1996.

[81] D. E. Long. *Model checking, Abstraction and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.

[82] Abadi M. and Lamport L. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[83] Panagiotis Manolios and Sudarshan K. Srinivasan. Refinement Maps for Efficient Verification of Processor Models. In *DATE*, pages 1304–1309, 2005.

[84] R.C. Martin. Design Principles and Design Patterns. Microsoft, `www.objectmentor.com`.

[85] K. L. McMillan and J. Schwalbe. Formal Verification of the Encore Gigamax Cache Consistency Protocols. In *International Symposium on Shared Memory Multiprocessors*, pages 242–251. Information Processing Society of Japan, April 1991.

[86] Kenneth L. McMillan. Cadence SMV. Available from `http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/smv/`.

[87] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[88] Kenneth L. McMillan. A Compositional Rule for Hardware Design Refinement. In *CAV*, pages 24–35, 1997.

[89] Kenneth L. McMillan. Getting Started with SMV. March 1999.

[90] Kenneth L. McMillan. The SMV Language. March 1999.

[91] Sun Meng, Zhang Naixiao, and Luís Soares Barbosa. On Semantics and Refinement of UML Statecharts: A Coalgebraic View. In *SEFM*, pages 164–173, 2004.

[92] M.Fujita. Debugging a Communications Chip. IEEE Spectrum 33, 6 (June 1996): 64.

[93] L. Millett and T. Teitelbaum. Slicing PROMELA and Its Applications to Model Checking, 1998.

[94] Lynette I. Millett and Tim Teitelbaum. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *STTT*, 2(4):343–349, 2000.

[95] Robin Milner. An Algebraic Definition of Simulation Between Programs. Technical report, Stanford, CA, USA, 1971.

[96] Object Management Group (OMG). Unified Modeling Language Specification. Available from `http://www.omg.org/uml/`.

[97] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

[98] Girish Keshav Palshikar. An Introduction to Model Checking. Available from `www.eetasia.com/ARTICLES/2005FEB/B/2005FEB16_EMS_ST_TA.pdf`, February 2005.

[99] Ivan Paltor and Johan Lilius. Formalising UML State Machines for Model Checking. In *UML*, pages 430–445, 1999.

[100] Larry Paulson and Tobias Nipkow. Isabelle. Available from `http://isabelle.in.tum.de/`.

[101] Doron Peled and Thomas Wilke. Stutter-Invariant Temporal Properties are Expressible without the Next-Time Operator. *Information Processing Letters*, 63(5):243–246, 1997.

[102] Quantum-Leaps. Brief Introduction to UML State Machines. `http://www.quantum-leaps.com/devzone/cookbook.htm#State`.

[103] Terry Quatrani. Introduction to the Unified Modeling Language, 2003.

[104] J. Rumbaugh. State Machine View. `www-128.ibm.com/developerworks/rational/library/content/04August/3153/3%153_Rumbaugh_ch07.pdf`.

[105] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., 1999.

[106] Shane Sendall. *Specifying Reactive System Behavior*. PhD thesis, 2002.

[107] Ian Sommerville. *Software Engineering*. Addison Wesley Professional, 6th edition, 2000.

[108] Tirumale Sreemani and Joanne M. Atlee. Feasibility of Model Checking Software Requirements. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 77, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[109] SRI. Pvs Specification and Verification System. Available from `http://pvs.csl.sri.com/`.

[110] Sparx Systems. `http://sparxsystems.com.au/EAUserGuide/index.html?activeclasses.htm`.

[111] Tigris. Argo. Available from `http://argouml.tigris.org/`.

[112] F. Tip. A Survey of Program Slicing Techniques. *Journal of programming languages*, 3:121–189, 1995.

[113] Issa Traoré. An Outline of PVS Semantics for UML Statecharts. *J. UCS*, 6(11):1088–1108, 2000.

[114] STL Queen's University. References: Semantics of UML State Machines (Statechart Diagrams). Availble from `http://www.cs.queensu.ca/home/stl/internal/uml2/bibtex/ref_umlstatemach%ines.html`.

[115] S. Van Langenhove and A. Hoogewijs. Integrating Cadence SMV in the Verification of UML Software. In *Proceedings of the 8th Dutch Proof Tools Day*, pages 15–29. Foundations group of the NIII, Nijmegen, The Netherlands, July 2004.

[116] S. Van Langenhove, A. Hoogewijs, and B. De Leeuw. UML based Verification of Software. In *Proceedings of the 32nd Spring School in Theoretical Computer Science, Concurrency theory and Applications*. CIRM, Luminy, Marseille, France, April 2004.

[117] Sara Van Langenhove. Internal Broadcasting to Slice UML State Charts: As Rich As Needed. FNRS Contact Day: The Theory and Practice of Software Verification, October 13th, 2005 - Liège, Belgium.

[118] Sara Van Langenhove. Towards an Optimal Slicing of Hierarchical Automata. CS Seminar, July 1st, 2005 - Amsterdam, The Netherlands.

[119] Sara Van Langenhove. UML-Based Approach to Developing Verified Embedded Software. Proceedings of the 3th PhD FWET Symposium, 2005.

[120] Sara Van Langenhove. Protocol Conformance through Refinement Mappings in Cadence SMV. In *Proceedings of the Joint BeNeLuxFra Conference in Mathematics*, pages 50–51, May 2005.

[121] Sara Van Langenhove. Protocol Conformance through Refinement Mappings in Cadence SMV. *To appear in the Bulletin of the Belgian Mathematical Society (BMS)*, 2006.

[122] Ann Vanhoof. GUI als Hulpmiddel bij de Ontwerpverificatie van UML Software. Master's thesis, Hogeschool Gent, Departement Industrièle Wetenschappen, 2005.

[123] Dolores R. Wallace, Laura M. Ippolito, and Barbara Cuthill. Reference Information for the Software Verification and Validation Process. NIST Special Publication 500-234, March 1996.

[124] Ji Wang, Wei Dong, and Zhi-Chang Qi. Model Checking UML Statecharts. *Eigth Asia-Pacific Software Engineering Conference (APSEC'01)*, 2001.

[125] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing Hierarchical Automata for Model Checking UML Statecharts. *Lect. Notes Comp. Sci.*, 2495:435–446, 2003.

[126] Mark Weiser. Programmers Use Slices when Debugging. *Commun. ACM*, 25(7):446–452, 1982.

[127] Mark Weiser. Program Slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[128] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, 1979.

[129] Whatis.com. `http://whatis.techtarget.com`.

[130] Robin J. Wilson. *An Introduction to Graph Theory*. Longman, 1996.

[131] Terry Winograd, editor. *Bringing Design to Software*. ACM Press, 1996.

Abbreviations

A

AC(t)	Actions of transition t	48
ADT	Abstract Data Type	137
AP	Atomic Propositions	58

B

BDD	Binary Decision Diagram	229
BSM	Behavioral State Machine	136

C

CaSMV	Cadence SMV	20
CFG	Control Flow Graph	156
CTL	Computation Tree Logic	126
CTL_X	Computation Tree Logic without the next-time operator	173
CV(t)	Set of referenced variables in the guard of transition t	161

D

DEF(n)	The set of variables defined (assigned) at node n	157
Digraph	Directed Graph	180
DV	Set of update variables	160

E

EEG Environment Event Generator 52

EHA Extended Hierarchical Automaton 46

ES Extended set of states; $\in$ same automata as NS and NT 165

ET Extended set of transitions; $\in$ same automata as NS and NT 165

EV(t) Trigger Event of transition t 48

F

FIFO First In First Out 44

FSM Finite State Machine 34

G

G(t) Guard of transition t 48

GE Set of generated events 160

I

IS Important set of states to find the dependent elements 165

IT Important set of transitions to find the dependent elements .. 165

L

LIFO Last In First Out 137

LTL Linear Time Temporal Logic 127

LTL_X Linear Temporal Logic without the next-time operator 172

M

MAE_x Mutilated Automaton Ending in x 184

MAS_x Mutilated Automaton Starting in x 184

N

NS New set of states on which IS and IT depend 165

NT New set of transitions on which IS and IT depend 165

O

OMG Object Management Group 7

P

PDD Parallel Data Dependence 163

PDG Program Dependence Graph 157

PSM Protocol State Machine 136

R

$Refine_R$ A boolean refinement function to keep actions in final slice .. 165

RCD Refinement Control Dependence 164

RDD Refinement Data Dependence 164

REF(n) The set of variables references at node n 156

RS Result set of states to be reserved in the final slice 165

RT Result set of transitions to be reserved in the final slice 165

RTC Run-To-Completion 42

S

SRC(t) Source of transition t 48

SD Synchronization Dependence 163

SDD Sequential Data Dependence 161

SR(t) Source Restriction (real sources) of transition t 48

T

TCD Transition Control Dependence 163

TD(t) Target Determinator (real targets) of transition t 48

TE Set of triggering events 161

TGT(t) Target of transition t 48

TLA Temporal Logic of Actions 149

U

UML Unified Modeling Language 7

UV Set of referenced variables 160

X

XMI XML Metadata Interchange Format 62

XML Extensible Markup Language 62

List of Figures

List of Code Listing

List of Algorithms

List of Tables